IMPLEMENTATION OF THE ALGRES TESTBED (YEAR-2 VERSION)

Summary: This document describes the implementation of the ALGRES Testbed of Chimera (Year 2).

Author(s): Stefano Ceri, Stefano Crespi-Reghizzi, Piero Fraternali, Giovanna Guerrini, Gianfranco Lamperti, Stefano Paraboschi, Giuseppe Psaila

Contributor(s): Paola Abbi, Annalisa Apicella, Andrea Bassani, Davide Berveglieri, Antonio Bossi, Carlo Cappellini, Stefano Castangia, Lanfranco Colella, Paola De Prati, Pier Paolo Merli, Roberto Penzo, Paolo Rossetti, Daniele Roda, Renato Ronconi

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

Authorised by: Willem Jonker

Issue number: 1

Status: DEFINITIVE

Date of issue: 26.5.1994

Reference: IDEA.DE.3P.009.01

Number of pages: 87

CONTENTS

1 INTRODUCTION

1.1 Design Principles
1.2 Project Organization
1.3 Architecture of the Year-2 ALGRES Testbed
1.4 Previous Documents on Algres Testbed Implementation
1.5 Organization of this report

2 COMPILERS

2.1 Federal Compilation
2.2 Modules of Chimera
2.3 Compilers Organization
2.4 Type Checking
2.5 Conclusion

3 TRANSLATION

3.1 Intermediate Graphical Language (IGLOO)
3.2 Examples of IGLOO Representations
3.3 Data Dictionary Management for Triggers
3.4 Organization of the Translation Process
3.5 Computation of Event Instances
3.6 Translation of Procedural Expressions
3.7 Translation of Declarative Expressions
3.8 Examples of Translation

4 RUN TIME SUPPORT

4.1 Control Flow of the Run Time System
4.2 Execution of Executable Entities
4.3 Event Handling
4.4 Active Rule Processing
4.5 Event Formula Evaluation

5 USER-FRIENDLY INTERFACE

5.1 Global Architecture of the UFI
5.2 Data Definition Interface
5.3 Rule and Transaction Interface
5.4 Future Development

6 ALGRES++

6.1 Blocks and control structures
6.2 Conditional Expression

6.3 Subprograms
6.4 Links

7 CONCLUSIONS

A1 GRAMMAR OF THE YEAR-2 ALGRES TESTBED

A1.1 DD Module
A1.2 TF Module
A1.3 QUAC Module
A1.4 Update Primitives (UP Module)
A1.5 TRIG Module
A1.6 DML Module

1 INTRODUCTION

Contributed by: S. Ceri, S. Crespi Reghizzi

This document is a description of the ALGRES testbed implementation developed at the end of the second year of the IDEA Project; due to coordination with ECRC, the ALGRES testbed was focused (since the beginning of year 2) on the support of active rules of Chimera, while passive rules are the main focus of ECRC's ECLiPSe testbed. The ALGRES prototype is based on the document *Consolidated Specification of Chimera* [1], and is influenced by the architectures proposed for the industrial prototype of IDEA [2,4] and from the architecture of the ECLiPSe testbed [3].

1.1 Design Principles

The architecture of the ALGRES testbed is motivated by the following design principles.

- The ALGRES testbed should provide an implementation of triggers of Chimera that enables the evaluation of their functionalities from an application-oriented perspective.
- The implementation of the ALGRES testbed should provide insight on all issues related to trigger management so as to enable their technology transfer to ADS [20].
- In order to ease the technology transfer to a different database engine, the peculiar features of the ALGRES platform used for rapid prototyping should be as much as possible encapsulated.

The above design principles have been translated into an architecture which is clearly separated into four parts:

- A *virtual interface* (VICHI) which interacts with ALGRES in accessing and manipulating data dictionary components and provides generic services to the compilers.
- Several *language compilers* which perform the translation from Chimera primitives into calls to the VICHI services.
- A *run-time system* responsible for activating and managing all computations, interacting with compilers or VICHI when necessary.
- A *user-friendly interface* which enables the interactive presentation of schema definitions (types, classes and triggers) and transactions; the interface is particularly suited to monitor and debug the execution of active rules.

In order to enhance reusability of the implementation, the ALGRES testbed has been designed and implemented with an *object-oriented approach*. In particular, implementation was based on the use of (reusable) classes and of their services, specified and subsequently implemented in C++.

The ALGRES testbed is designed on top of ALGRES [5], an extended relational database supporting complex nested relations and a powerful relational algebra, particularly suited to the rapid prototyping of advanced database architectures [6]; ALGRES was enhanced in the course of the IDEA project so as to better serve its requirements. We are currently designing the ALGRES testbed aiming at encapsulating all ALGRES-dependent features within the virtual interface (VICHI); in this way, compilation techniques are more easily testable and portable.

1.2 Project Organization

The design and implementation of a complex innovative language like Chimera is a challenge, and the difficulty is compounded with the short duration of the IDEA project. Such time constraints prevent the traditional schedule consisting of first specifying the language, then implementing it. Quite opposite to that, we were forced into a "simultaneous engineering effort", with many project teams working on different but interrelated aspects, ranging from language specification, to compiler design and run-time support organization.

The very tight project scheduling motivated our search for an approach that would minimize the impact of changes and extensions. Modularization and good interfacing of sub-systems is of course the key to reducing the consequences of unplanned change as well as of planned extensions. In addition to that we relied on compiler generating tools, also specially developed to satisfy our needs. Planned extensions are of course the source of macroscopic changes: for instance the year 1 prototype was restricted to data definition and the richest language features of active rules have been fully specified and added in year 2.

Algres itself has been significantly revised in years 1 and 2 to better match the requirements of Chimera as well as to improve run-time efficiency.

The ALGRES Testbed project has so far been carried out by 19 master-level students doing their thesis for achieving the Engineering Degree at the Politecnico di Milano. The following groups were active during the first two years of IDEA:

- **Schema compiler** (DD module): year-1 development and year-2 maintenance (2 students).
- **Trigger and Transaction Compilers** (TRIG and TRANS modules) (3 students);
- **VICHI year-1** library (3 students);
- **VICHI year-2** library (2 students);

- **Code generation** for active rules and transactions (2 students);
- **Run-time support** (3 students).
- **User-friendly interface** (2 students);
- **Application Support (in cooperation with TXT)** (2 students).

1.3 Architecture of the Year-2 ALGRES Testbed

The ALGRES Testbed is developed on Sun workstations running the SUN-OS 4.1.3 Operating System; the code is written in GNU C++ and uses the GNU standard libraries; the user interface is implemented in *Motiv*. Inter-processes communication is based on a client/server architecture, where the user-friendly interface acts as client process and activates, in turns, several server processes which are responsible for compilation and execution. In turn, compilers and run-time support play the role of clients wrt the ALGRES database system, which runs as a separate process. Figures 1, 2, 3, and 4 indicate the process organization and main interprocess communications which take place during the processing of Chimera statements. Processes are enclosed into boxes with rounded corners, while system components are enclosed into boxes with sharp corners.

Execution is monitored by the user-friendly interface (UFI), which has a modular organization. The driver module of the user-friendly interface may act also as a simple application program interface (API) specifically tailored to the needs of the TMS application, developed at TXT.

Figure 1 illustrates the behavior of the system when schema definitions are processed (Year-1 testbed); the main data and control flows are numbered in the figure. When a schema is presented by a user for compilation, a module of the user-friendly interface, called UFI-DD, activates the DD compiler; Chimera definitions are processed by the compilers one at a time (communicated along a pipe in input to the compiler, 1), and error messages or termination are returned to the UFI-DD process (along another pipe in input to UFI-DD, 4). The DD compiler uses VICHI services (2) for reading the current schema, described by the Data Dictionary (3.1), and produces changes to the Data Dictionary (3.2) as well as the schema definitions (3.3) which are used in order to create the ALGRES database.

Figure 2 illustrates the similar behavior when trigger definitions are processed; in this case, UFI-DD activates the TRIG compiler. In addition to accessing the Data Dictionary (3.1), VICHI provides to the TRIG compiler services for producing an internal representation of triggers (called IGLOO representation: flow 3.2) which is next used by the code generator (3.3) in order to produce Algres code for triggers (3.4) and to store the relevant information about triggers into the Trigger Dictionary (3.5).

Figure 3 illustrates the behavior of the system when a transaction line is presented. Transactional commands, such as *open-db, close-db, begin-transaction, commit,* and *rollback*,

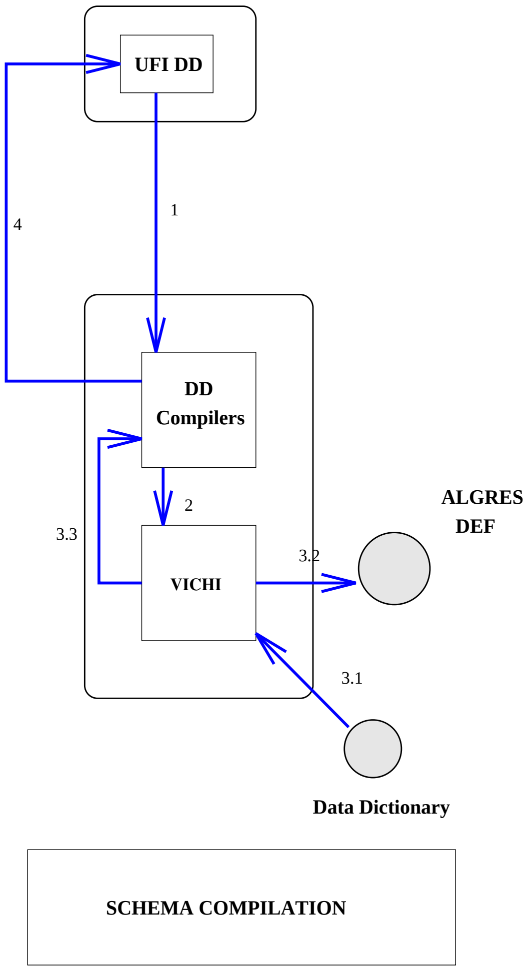

Figure 1: Schema Compilation

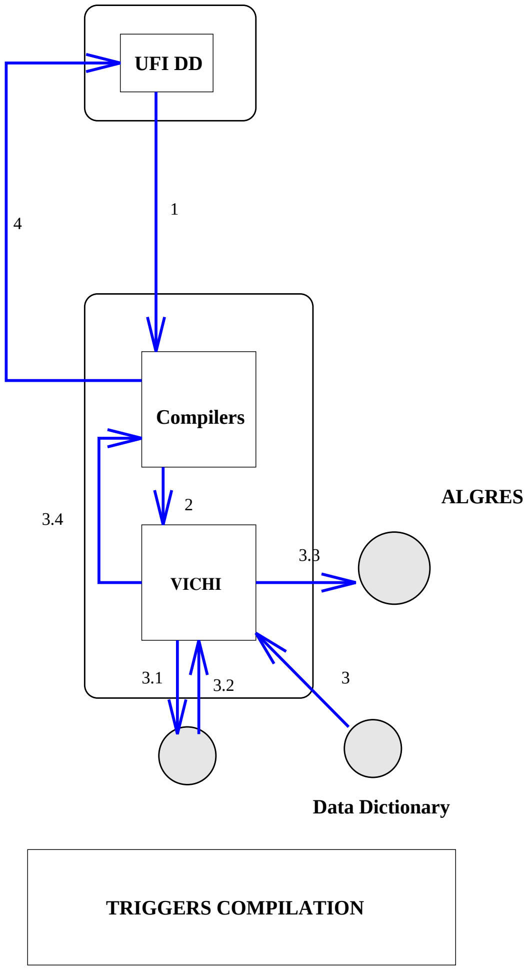

Figure 2: Trigger Compilation

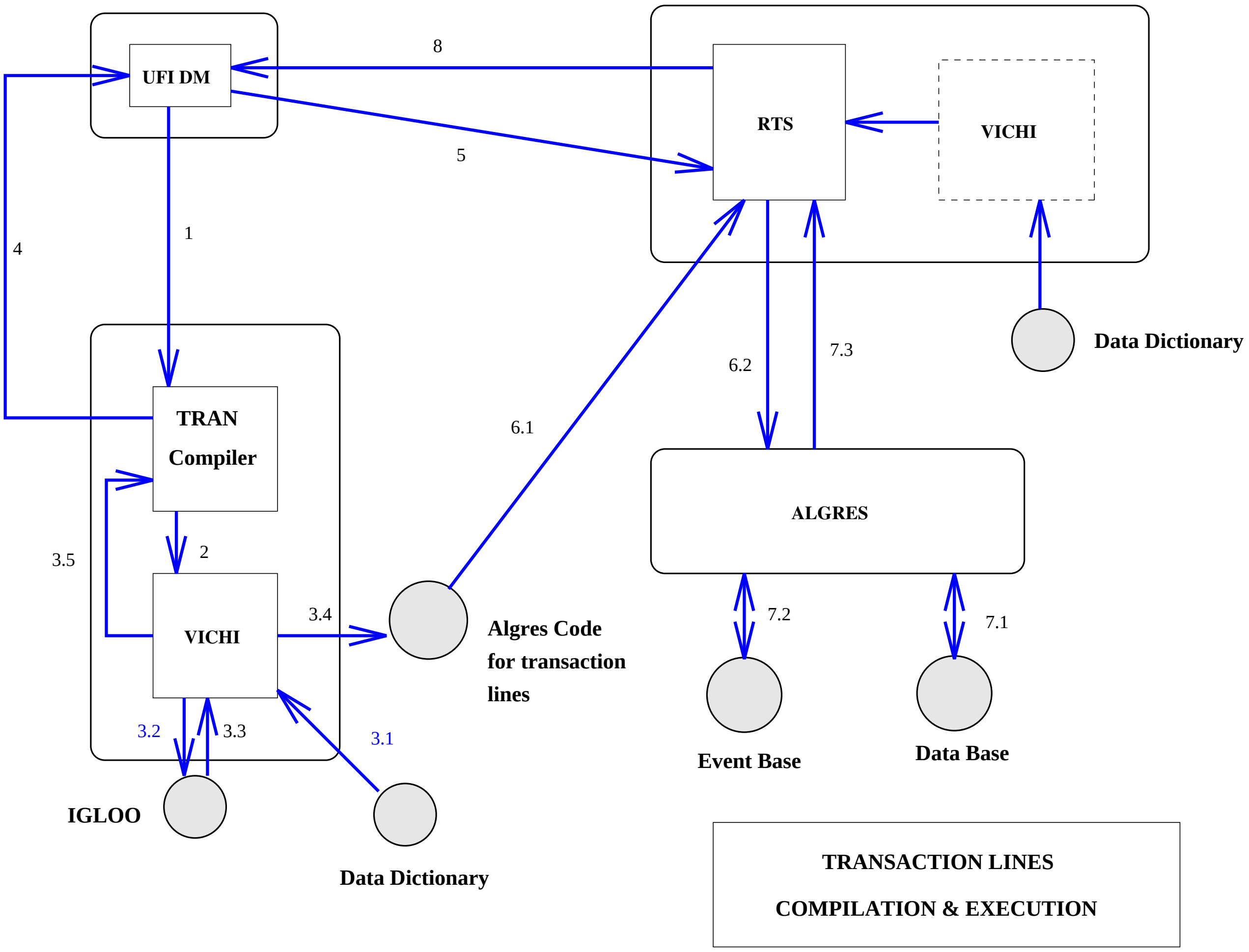

Figure 3: Transaction Line Compilation and Processing

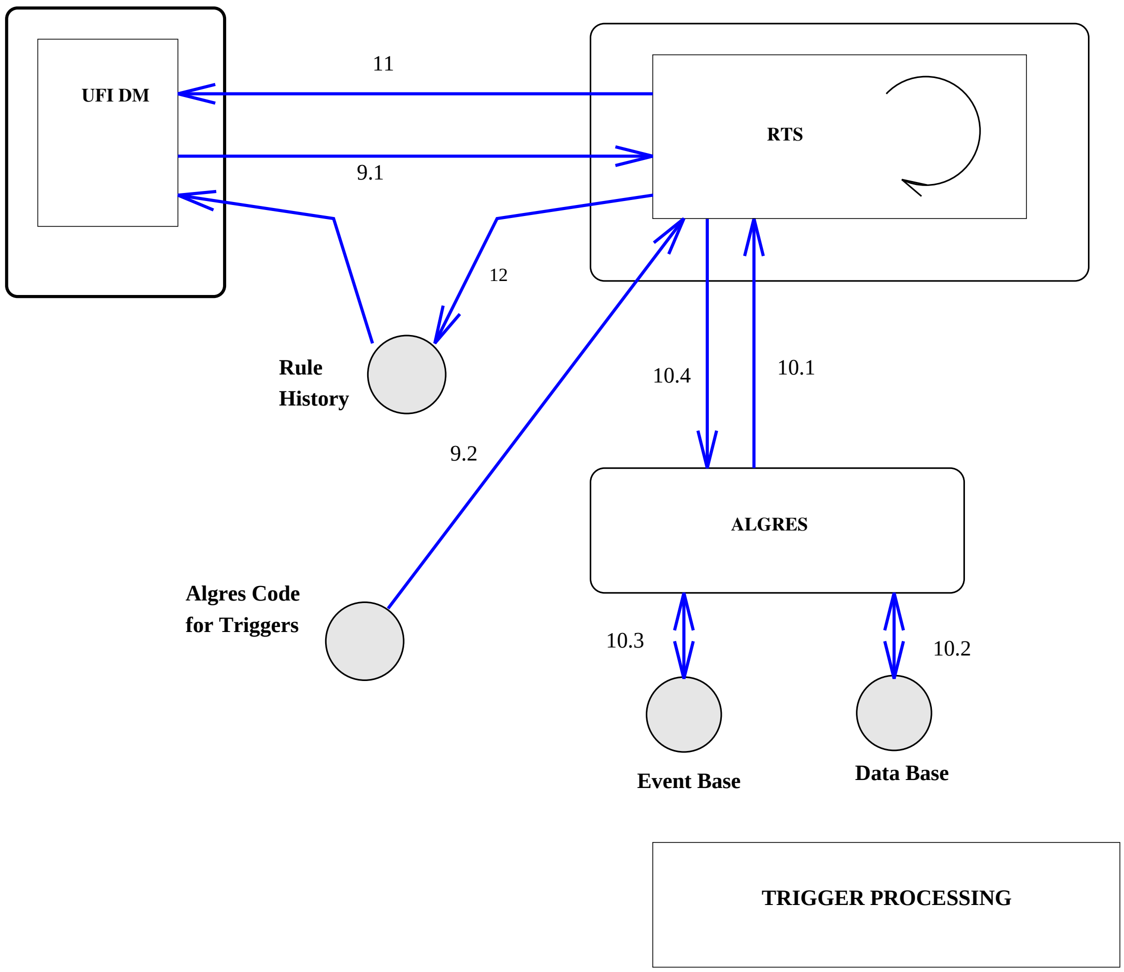

Figure 4: Trigger Processing

are directly executed by the run-time process. In particular, the *open-db* command causes the initial loading of a limited amount of information from the Data Dictionary (by means of few VICHI services directly linked to the run-time process); this information is used for the run-time checks which are not performed by compilers.

The ALGRES testbed exhibits a *compile-and-go* approach: transactions lines are compiled by producing ALGRES code which is next executed and removed. When a user transaction is presented, a different module of the user-friendly interface, called UFI-DM, submits the transaction line to the TRAN compiler, which executes the compilation in a way that is very similar to the compilation of a trigger; in particular, ALGRES code for the transaction line is produced and stored into a file. When compilation is completed (4), UFI-DM activates the run-time support process (5), which in activates the ALGRES process (6.2) which executes the program produced by the compiler (6.1). ALGRES modifies the database (7.1) and the eventbase (7.2), thus preparing for trigger processing. Finally, the run-time process returns the control to the UFI-DM process.

Figure 4 illustrates the behavior of the system when trigger processing is activated. Triggers can be executed either in a *continuous mode*, without interrupts until the fixpoint is reached, or in a *debugging mode*, which enables step-by-step control. In this case, the execution of the triggers is driven by an interactive interface, which allows the user to halt the execution of triggers, inspect the past trace and the actual state of execution, and even intervene on the conflict resolution strategy, to direct rule processing. When rule triggering is activated (at the end of each transaction line or after a *commit* or *savepoint* command: flow 9), the run-time system determines the first trigger to be fired, if any, based on occurred events and on trigger priority. Then, execution continues by activating the ALGRES process (10.2) and submitting to algres the trigger code (10.1); trigger execution consists of evaluating the condition first, then the action); the evaluation uses the information stored in the database (11.1) and eventbase (11.2). Prior to returning the control to the UFI-DM module (12.2), an execution history is produced (12.1) which enables a possible analysis and tracing of trigger processing.

1.4 Previous Documents on Algres Testbed Implementation

This document presents a general description of the Algres Testbed architecture and is self-contained; however, several other documents complement this one and can be used as a reference, including:

- A general description of the ALGRES Testbed (Idea Report DE.3P.003.02)
- Documents describing the Year-1 implementation, in particular: "Specification of DD Compiler" (Idea Report WP.3P.004), "Specification of Vichi-DD" (Idea Report WP.3P.005), and "Specification of the Interface between Vichi-DD and the DD-Compiler" (Idea Report WP.3P.006).
- Design documents concerning the Year-2 testbed, in particular: "Specification of the UFI and trigger Compilers for Chimera (Algres testbed)" (IDEA Report DE.3P.008.01) and "Specification of the run-time support for Chimera (Algres testbed)" (IDEA Report DE.3P.007.02).

1.5 Organization of this report

- Section 2 is focused on the modularization of compilers and on the description of the year-2 compilers (TRIG and TRANS).
- Section 3 describes the intermediate representation used for Chimera statements, called IGLOO, and describes the translation algorithms used for generating ALGRES code from IGLOO representations
- Section 4 describes the run-time system and indicates how triggers are executed
- Section 5 describes the user-friendly interface and the various options for interaction (including step-by-step monitoring and debugging)

- Section 6 describes enhancements to Algres that were done in the context of the IDEA project.

2 COMPILERS

contributed by: C. Cappellini, R. Penzo, G. Psaila

Compilers which have been designed in the Algres testbed implementation of Chimera have suffered several language changes, concerned the syntax and semantic levels. Such changes have been rather frequent due to the innovative concepts of Chimera and the complex interactions of aspects like inheritance, triggering, schema evolution. Language revisions, in agreement with IDEA partners, have been intense in year 1 and beginning of year 2, then slower; but minor changes and refinements are still occurring. This situation has been mastered in our project thanks to the use of Grammatica, a high-level tool for compiler generation.

Subsystems separation was designed into the prototype by partitioning the language into the Data or Conceptual model (DD compiler), the Active rules model (TRIG compiler) and the Transaction model (TRANS compiler). In order to allow a finer decomposition of sublanguages we exploited the Federal Grammars approach [15], a new technique for decomposing a complex syntax and attribute grammar into non-mutually recursive modules. By this technique most language changes are circumscribed to a single module. The compilers have been specified using attribute grammars and implemented by the compiler generation tools Grammatica-ATELIER. Semantic functions are supported by the VICHI libraries of year-1 and year-2.

2.1 Federal Compilation

We call "Federal Compiler" a modular compiler obtained by aggregating several syntactic and semantic modules cooperating together. The main distinction between this approach and conventional software modularization is that the modularization is decided at the level of grammar productions; consequently, it is possible to manage the interconnection of compilers within the environment for compiler construction, that automatically generates and maintains the compilers from high-level specifications [16].

In designing the grammar modularization, there is only one restriction: the modules must call each other in such a way that the calling scheme be a directed acyclic graph. Each **syntax module** (or **s-module**) of a federal grammar exports one or more nonterminal symbols, called **sub-axioms**, and can import other nonterminal symbols exported by one or more of the other s-modules; such a mechanism defines an import-export relationship between modules that must be acyclic, so that the result of the decomposition process is a directed acyclic graph. The axiom of the global grammar (obtained by joining all the s-modules) is the axiom of the **top module**, an s-module which imports nonterminals from the other modules but whose exported nonterminal (the axiom) is not imported by any other module. If there is more than an s-module, the graph defines more than language, one for each top-module in the graph; these languages are different but they have common parts, for example exepression, formulas, and so on.

An s-module is completed by means of semantic rules associated to its productions: Grammatica uses the well known **Attribute Grammar** technique to describe semantics. The semantics of an s-module imports attributes associated to the imported nonterminals and exports attributes associated to its exported nonterminals, while the characterizing concepts of the Attribute Grammars do not change.

Federated grammars provide several advantages:

- They enforce modular design of the language grammar, which is therefore carefully structured so as to match significant compilation units.
- They promote the reuse of *s-modules* which are called from several other *s-modules* (for instance, in the Year-2 subset, the TF-Module is used by TRIG, QUAC and UP Module; clearly, TF-Module will be reused also in the Year-3 and Year-4 testbeds).
- They allow several design teams to work on distinct parts of the whole grammar; each team's work is quite indipendent from the other teams. This "divide-and-conquer" technique lowers the complexity of design and implementation.
- Modularization of federated grammars in the context of the Grammatica tools provides several advantages from a software engineering perspective, such as information hiding, programming via abstract specification languages, and automatic maintenance of compilers; these features are very beneficial in projects which have a long development and a fast turn over of developers, such as the Algres testbed of Chimera.

2.2 Modules of Chimera

Chimera is a rich language; thus, it is possible to separate the compilation process into several components:

1. DML-Module : Top Module for the Data Manipulation Language (transactions).
2. DD-Module: Module of the Conceptual Model (schema elements).
3. TRIG-Module: Top Module for Trigger implementation.
4. QUAC-Module: Module of Query and Action Primitives.
5. UP-Module: Module of Update Primitives.
6. TF-Module: Module of Terms and Formulas.

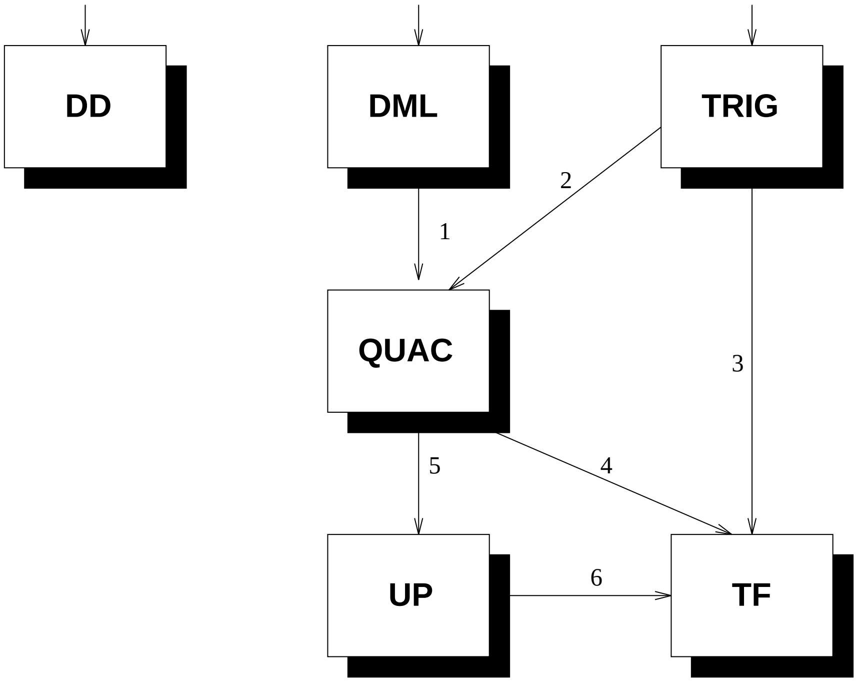

Figure 5: DAG of *s-modules* in the Grammar for Year-1 and Year-2 Algres Testbed of Chimera. Other s-modules remains to be added

7. PR-Module: Module of Passive Rules (including: attributes and views, not implemented).
8. OP-Module: Module of Operations (not implemented).
9. CO-Module: Module of Constraints (not implemented).

Each component is associated to a specific subset of the productions in the syntax of the Chimera language, as specified in the document [1]. In the Appendix 1 we show the grammar modules obtained by applying the technique for modularizing the compiler construction process; this technique has been developed by S. Crespi-Reghizzi and G. Psaila, reported in [15]. Appendix 1 also illustrates the grammar productions which are accepted by the Year-2 testbed.

Figure 1 illustrates the import-export DAG of *s-modules* of the Chimera Grammar for the Year 1 and Year 2 Algres Testbed; arcs are directed from the importing module to the exporting module and are labelled with a number; these numbers refer to the table, reported in Figure 2, listing the nonterminals in the actual grammar productions which are exchanged between each pair of connected *s-modules*.

2.3 Compilers Organization

The grammar modules currently supported by the Year-2 testbed have been assembled in three different way, each of which correspond to a different executable compiler. That

Arc	Nonterminals	Export	Import
1	ACTION0	DML	QUAC
2	ACTION0	TRIG	QUAC
3	ATOMIC_FORMULA0	TRIG	TF
4	EXPLIST0 START_WITH_IDENTIF0 START_WITH_VARIABLE0 FORMULA0	QUAC	TF
5	UPDATE0	QUAC	UP
6	RECORD0 EXPRESSION0	UP	TF

Figure 6: Import-Export Relationship in the Grammar for the Year-1 and Year-2 Algres Testbed of Chimera

means we do not have a unique compiler for the Chimera language; we have three different compilers, each one specialized in a particular section of the language. The main advantage of such an approach is the incremental building. as side effect we have got a strong separation among different phase of programming in Chimera; that is a desirable consequence, because they differs w.r.t. the context they operate in (e.g. data definition operation are logically different from transaction execution).

2.3.1 DD Compiler

The DD (Data Definition) Compiler is targeted on the Class definition section of the Chimera language. It has been developed for the Year-1 prototype and is based on the grammar module called DD. Its work consists of compiling new class schemas, collecting them into the *Data Dictionary* and generating the corresponding data structures in the repository. The interaction with our database machine (ALGRES) is encapsulated in the VICHI Library (Virtual Interface for CHImera) developed for the Year 1 protorype (then the library is shortly calles Vichi-1).

2.3.2 TRIG Compiler

The TRIG Compiler has the task of compiling Triggers and Active Rules. It is based on the grammar modules TF (Terms and Formulas), UP (Updates), QUAC (Query and Actions) and TRIG (the entry point to the Trigger sublanguage). Every section in a trigger definition uses Terms and Formulas, while Updates and Queries can be found in the Action part of a trigger.

This compiler substantially acts like the DD compiler, because it puts triggers into the Rule Dictionary; but it also translates the trigger into the target language, that is the programming language of our database machine (ALGRES). To do that it includes and uses the extended version of the VICHI library, developed for the Year 2 prototype (then the library is called Vichi-2).

2.3.3 TRANS Compiler

The task of the TRANS (Transaction) Compiler is to compile transactions, providing the corresponding code which is immediately executed by the Runtime System. This is the difference between this compiler and the other two ones : the TRANS compiler must dialog in a more complicated way with the UFI and it also has to communicate its results to the Runtime system. The context in which it works is then very different and is characterizing of it. This compiler is based on the grammar modules TF (Terms and Formulas), UP (Updates), QUAC (Query and Actions) and DML (the entry point to the data manipulation sublanguage). Many of them are used in the TRIG Compiler as well, because they provides parts of the Chimera language which are common to the whole language (e.g. formulas, or updates). For the rest it follows the same organization of the TRIG Compiler, based on Vichi-2.

2.4 Type Checking

Chimera is a typed language; as consequence, extensive static type checking control should be done during the compilation phase. Certainly it is not possible to check all statically, because Chimera is an object oriented database language which provides inheritance and object migration, but most of the controls can be done during compilation.

The type checking process is organized as follows :

- Type definition (during class definitions' compilation).
- Variable declarations' analysis and instructions' analysis (during trigger compilation and transaction compilation).

We recall that in Chimera there are three type categories:

- Predefined Types
  Predefined types are integer, real, boolean, char, string. There also exist the *null* type (although not implemented in the Year-2 Algres prototype), that is always compatible with every other type, and the OID type, corresponding to an Object IDentifier.
- User Defined Types
  They are built by the user applying the type constructors *set*, *list* and *record*, by means of which is possible to obtain arbitrarily complex structures. Notice that in Chimera it is not possible to introduce non-structured user-defined types.

- Class Types
  They are the types implicitely introduced with object class declarations. From the type checking point of view they are considered as records, whose fields are attributes described in the related class definition, both the explicit and the implicit ones (the latter ones deriving from inheritance and object management, like the *self* attribute).

The following compatibility rules have been implemented in the Chimera compilers:

- Compatibility between predefined types
  - The *integer* type is compatible with the *real* type and vice-versa for every operation.
  - The *string* type is compatible with the *string(n)* type (string having at most a length of n characters).
  - All the other predefined types are incompatible each other.
- Compatibility between user-defined types
  In this case the type checking follows the structural compatibility principle. Two type are compatible if:
  - they have the same structure,
  - the types of every sub-fields are compatible.

  These rules are recursively applied when structured sub-fields are encountered.
- Compatibility between class-types
  Two class-types are compatible if one class is ancestor of the other or if the two classes are the same class.
- Compatibility between predefined and user-defined types
  Predefined types are incompatible with user-defined types.
- Compatibility between predefined types and class-types
  Because a class-type is a record, it is incompatible with predefined types (that is due to the fact that at the moment value classes are not implemented).
- Compatibility between user-defined types and class-types
  This case is similar to the compatibility between user-defined types, except that a class-type is always a record; then the following conditions hold :
  - The user defined type must be a record;
  - Structural compatibility between the two types must hold.

  Notice that the record corresponding to the class-type contains fields for both the esplicitily declared attributes and the implicitily inherited attributes (as ancestors' attributes and the *self* attribute).

We observe that value-classes are not implemented yet. Their implementation would certainly complicate the previous taxonomy, because a value-class can be defined by means of type constructors (set, list and record).

The rules mentioned above are applied by the Type Checking System everywhere there must be type compatibility between two objects or some instruction or predicate requires a particular type for its parameter; in any case expressions or formulas or anything else which infringes the rules are not allowed and compilation fails.

2.5 Conclusion

In our opinion the overall organization of the prototyping effort was rather successful. In particular some advanced software engineering techniques were very effective. Object-oriented libraries (VICHI) provided good reuse of design and of components. Syntax-modularization (federal grammars) minimized the impact of language changes. Abstract attribute grammars decoupled the compiler from the object-oriented library. Algres confirmed its central role for rapid prototyping of data-intensive algorithms.

Compiler generating tools have reduced the effort to revise compilers after language changes and helped in maintaining consistency. Some extensions to the tools have been implemented, for allowing multi-sweep evaluation of semantic attributes. Looking at the man-power involved, our organization successfully integrated the work of some twenty individuals, including staff, graduate students and consultants.

As compilers were regenerated several times (about ten), some inconvenience was caused by the machine time (a few hours) for the rather complex TF module. Ongoing improvements to Grammatica-ATELIER should alleviate this trouble. To conclude, the orderly structure of the Algres prototype places the project in the position to face without fear the future extensions and revisions that a large innovative language always requires.

3 TRANSLATION

contributed by: P. Abbi, A. Apicella, A. Bassani, P. De Prati, S. Paraboschi

This section introduces the IGLOO representation for Chimera programs and next focuses on the generation of Algres programs from IGLOO representations.

3.1 Intermediate Graphical Language (IGLOO)

IGLOO is a graphical language used to represent CL, the Conceptual Language of Chimera [1]. We have chosen to define a new intermediate language, instead of directly tran-

[1]IGLOO is an acronym standing for Intermediate Graphical Language; the ubiquitous Object-Orientation-ness of Chimera allows us to complete the acronym.

slating from Chimera into Algres, because this intermediate language permits a simpler translation process, and enables us to focus on the optimization problems more easily.

Although IGLOO highlights a graphical representation, it is internally supported by means of a tabular representation; in particular, IGLOO is the language managed by VICHI-CL, and as such all intermediate representations written in IGLOO will be stored, by means of hash tables, in the memory-resident portions of the data dictionary.

IGLOO allows a clear identification of the partitioning of the work between the Compilers and the VICHI-CL teams: the former have the responsability of parsing input Chimera, type-checking CL statements, and translate them into IGLOO; the latter have responsibility of optimizing IGLOO representations and of translating them into ALGRES, thereby generating suitable ALGRES code. For porting our testbed to a different target (e.g., Flora), only the code-generation components of VICHI-CL need to be rewritten.

3.1.1 Requirements for IGLOO

The following requirements have driven us during the language design:

- *Ease of Graphic Representation*: a graphical language is a lot more flexible and easier to understand than a textual one. The transformations that have to be applied during the translation and the optimization have often a simple correspondent to graph operations and substitutions. At the moment, we are not considering to use this representation for user interaction; we think that the study of this language will provide useful information toward the implementation of a true graphical interface for Chimera.
- *Possible structuring by means of abstraction levels*: Chimera can be viewed as being procedural at a higher level and declarative at a lower level. Actions are executed in sequence, while conditions and expressions are declaratively defined. IGLOO distinguishes the different components, particularly the declarative and the procedural part, that need to be treated in completeley different ways.
- *Complete cover of CL Language*: The IGLOO language should enable us to describe any statement of CL, including active and passive rules, constraints, queries, and updates; further, it must support a notion of scope that permits to relate intermediate representations of correlated statements (e.g., all actions of a transaction line; the condition/action pair of a trigger; all the passive rules with the same head).
- *Conciseness*: The language should avoid redundancy and should not incorporate syntactic features that make it less readable. For this reason, we have excluded to use a language too close to syntactic trees. Indeed, grammar productions introduce a redundant hierarchy of levels (e.g., suppose of having a simple condition `true`, in the syntactic tree we have the sequence: `condition_formula` → `simple_condition_formula` → `pos_or_neg_atomic_formula` → `atomic_formula` → `expression` →

`sum_exp` → `mult_exp` → `term` → `simple_term` → `constant`; for every derivation we need a node in the syntactic tree). A particularly relevant feature that we need to support in IGLOO is the representation of bindings between variables. If a variable is used in different places, then the intermediate language must map it to the same element. This can be easily done with a graphical language, associating a unique element of the graph to the variable; the same does not apply to syntactic trees.

- *Appropriate level of abstraction for translation*: The language has to simplify the translation process, so it has to be intermediate between the source and the target language. It is particularly interesting the possibility of defining a translation as a set of small transformations on subparts of the graph.
- *Appropriate level of abstraction for optimization*: The intermediate representation must be suited to support several "generations" of optimization techniques, from the simple ones to the most sophisticated ones. This aspect is particularly relevant in the context of our research group, where the implementation is done by students who have a high turn-over.

3.1.2 Structure of IGLOO

Although the design of IGLOO will eventually include also passive rules and constraints, the present document is focused on the representation of the CL statements that will be accepted by the Year-2 Testbed compilers. Currently, the IGLOO language is being designed, therefore this document will only sketch examples of language elements that have been already designed; a complete description of the language is out of our current scope.

Given the above limitations, we focus on two "units of optimization and translation" **(UOT)** which are required by the Year-2 Testbed: **transaction lines** and **triggers**. UOT are constituted of CL statements; statements which are currently considered include:

- The condition of a trigger;
- Each individual action of a reaction;
- Each individual action of a transaction line.

Actions are Chimera query and update primitives, as identified by the Year-2 subset of the Chimera grammar.

The IGLOO language distinguishes three levels of representation of each UOT:

- *Level 1*: The higher level is used to represent the dependencies between all CL statements composing a UOT. At this level, each CL statement is represented

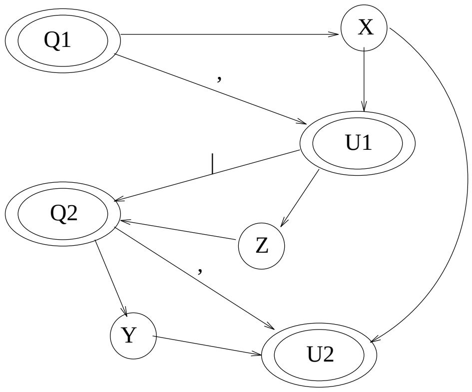

Figure 7: First level of IGLOO for a transaction line

by a node, labeled either as condition or as action; each variable shared between CL statement is also represented by means of a node. Directed arcs between CL statements identify their precedence relationship; these arcs are labeled by the kind of connector used: either sequence or pipeline. Directed arcs connect statement nodes and variable nodes, to represent which statements produce variable bindings and which statements use these bindings.

- *Level 2*: The intermediate level presents a refined level of abstraction; the goal of this level is to analyze the stratification of rules and formulas. In particular, we currently use level 2 for aggregate functions, which may include in their condition variables defined ouside the scope of the aggregate function itself; in such case, bindings for these variables need to be "passed" from the environment that encloses the aggregate function, thereby introducing a sort of stratification. Since we have no passive rules in Year-2 Testbed, we postpone the analysis of their stratification, but we anticipate that level 2 would be ideal for determining strongly connected components and strata.
- *Level 3*: This is the most detailed level, where all nodes corresponding to terms, expressions, and formulas have a suitable representation.

3.2 Examples of IGLOO Representations

We first present *generic examples* of levels 1 and 2; next we illustrate a transaction line at the three levels of representation.

3.2.1 First Level

An example of transaction line is illustrated in figure 7. We can observe two different kinds of nodes:

1. *Double Nodes*, representing statements

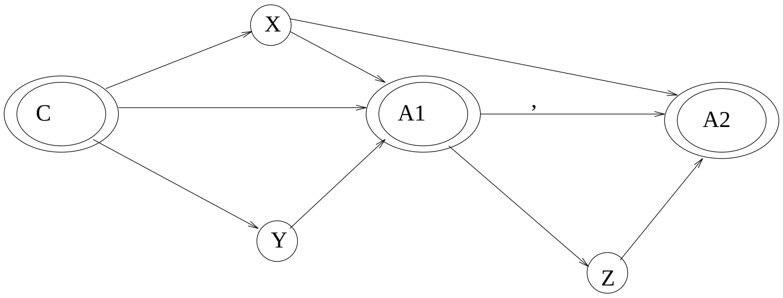

Figure 8: First level of IGLOO for a trigger

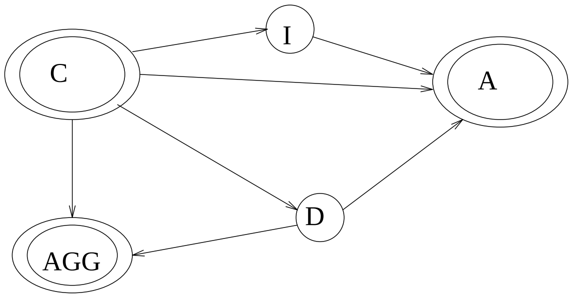

Figure 9: IGLOO second level for a trigger

2. *Simple Nodes*, representing variables shared among different statements

There are also two different kinds of directed arcs:

1. *Labelled arcs*, expressing the sequence of the statements. They can be labeled with a comma (",") or with a pipe ("|"), depending on the connector used in Chimera
2. *Simple arcs*, representing how variables are read and written

A condition for the correctness of the program is that the graph has to be acyclic.

An example of transaction line is illustrated in Figure 7. The line consists of a squence of query (Q1), update (U1), query (Q2), and update(U2). Q1's precedence wrt U1 is a sequence (indicated by a comma), while U1's precedence wrt Q2 is a pipeline (%), and Q2's precedence wrt U2 is again a sequence. Variable X is generated from query Q1 and is an input to updates U1 and U2. Variable Z is generated by U1 and used by Q2; variable Y is generated by Q2 and used by U2.

An example of trigger is illustrated in Figure 8. The condition corresponds to the first node, followed by a reaction consisting of a sequence of two actions A1 and A2.

3.2.2 Second Level

At this level, every statement is decomposed into the set of strata that have to be evaluated, giving evidence to the bindings between different strata. This is an aspect that will be particularly relevant in the future of the project, when passive rules will be translated. At the moment the problems associated with the stratification are rather simple, typically the only constraint is to consider clauses having aggregate functions only after having evaluated the clauses that may refer to the elements on which the aggregate function is computed.

In figure 9 we can see the second level graph corresponding to a trigger, where in the condition an aggregate function is used. The node labelled with AGG identifies an aggregate; the arc that connects it to variable D indicates that the aggregate function must receive bindings for that variable before its evaluation.

3.2.3 Third Level

In this level every node containing a condition or an expression is expanded to show its details. At every node of level 2 we correspond an entire graph. The meaning of nodes end edges is also different from the previous ones because here the context is declarative, instead of procedural. Nodes constitute a hierarchical structure:

- The top-level nodes represent user-defined classes that are used for defining the range of variables used in the expressions; built-in classes, which have no extension, are not represented in IGLOO.
- Nodes in the subsequent level introduce variables with an associated type; these nodes are linked to class nodes when a variable ranges over a user-defined class.
- Next, we have nodes corresponding to either attribute values, or constant values. Here, suitable node sequences may represent complex attributes or constants that can be defined in Chimera.
- Next, we have nodes representing comparison or membership operators; some of the arcs have an associated numeric label in order to distinguish the order of operand of noncommutative operations; e.g., arcs exiting from a "$\leq$" comparison operator.
- Finally, in the case of conditions, a node is used to represent a conjunction of predicates represented at the above level; thus, IGLOO embeds into its structure the conjunctive form of Chimera formulas.

In this graph nodes are represented by circles; a label associated with each node gives the type of the node. Examples 12, 13 and 14 illustrate specific examples of third-level IGLOO representations.

3.2.4 Example of Transaction Line in IGLOO

Consider the following transaction line in Chimera:

```
select(X where project(X),
               {[``Paolo'',''Rossi''],[``Marco'',''Bianchi'']} in X.members),


select(Y where project(Y), Y=X, integer(I),
                I=sum(Z.resourceavailability  where DBplanningperiod(Z),
                                                    Z.startdate='1/1/92',
                                                    Z.enddate='1/1/93'),
                integer(N), N=card(W where project(W), W=X),
                Y.resourceneeded>I/N),

modify(project.maxresourceallocable, Y, maxresourceallocable*1.1);
```

Figure 10 illustrates its IGLOO representation at level 1; Figure 11 illustrates its IGLOO representation at level 2; Figures 12, 13, and 14 illustrate the IGLOO representation of each individual CL statement at level 3.

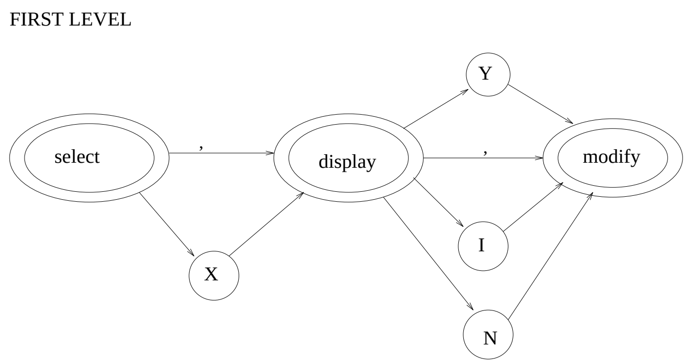

Figure 10: IGLOO graph of a transaction line

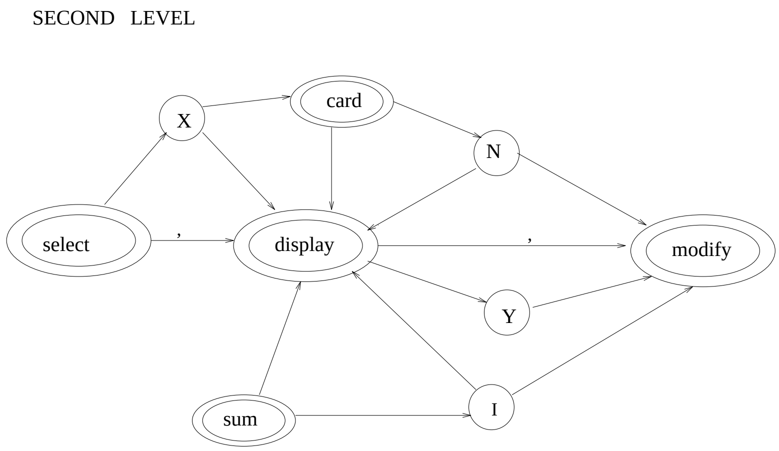

Figure 11: IGLOO graph of a transaction line

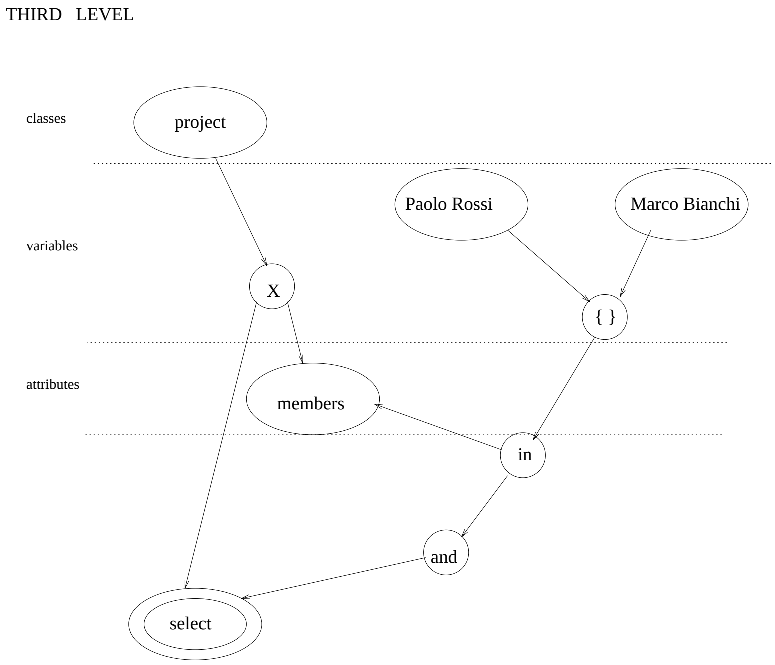

Figure 12: IGLOO graph of a transaction line

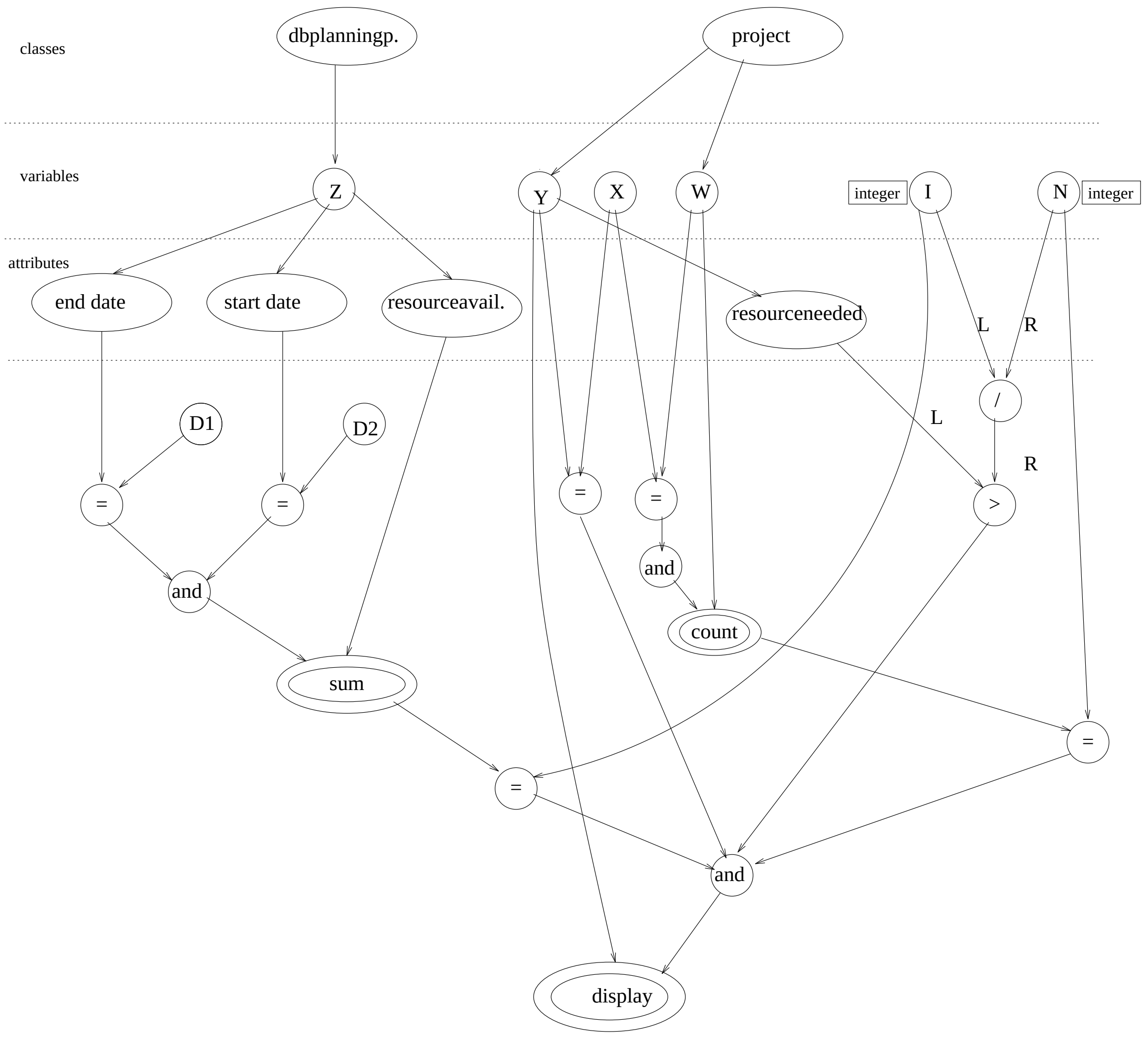

Figure 13: IGLOO graph of a transaction line

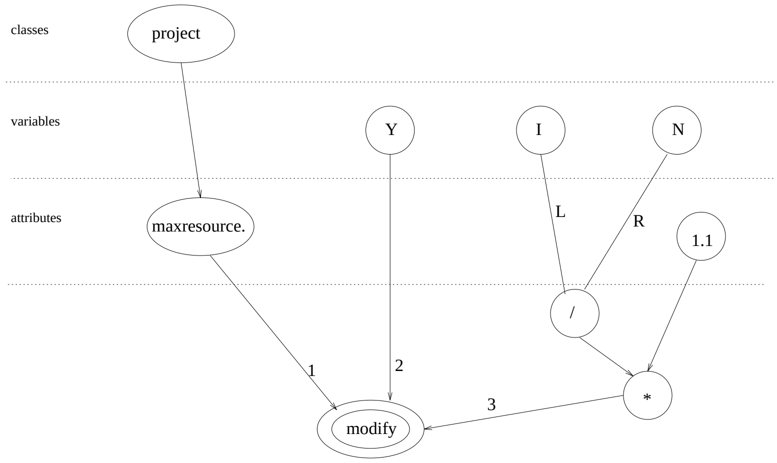

Figure 14: IGLOO graph of a transaction line

3.3 Data Dictionary Management for Triggers

VICHI-CL primitives are also responsible of updating the data dictionary for what concerns the description of triggers. *TIM* is an acronym for *Trigger Interface Manager*, a set of methods that enrich the services provided by Vichi-CM spoecifically designed for this purpose. The *Trigger Dictionary* is structured in two tables: (1) a table that contains most of the information regarding the trigger, (2) a table that contains the *Priority Graph*, a structure that represents all the priority relationships existing between triggers.

The main data structure of the *Trigger Dictionary* is an ALGRES table with the following schema:

Self Trigger identifier;

RuleName Trigger name;

Target if the trigger is targeted, the name of the class in which the trigger is defined, otherwise the null string;

Creator The user who has defined the trigger;

Consumption It is a flag that denotes the consumption mode (*true* if consuming, *false* if preserving);

Processing It is a flag that denotes the processing mode (*true* if immediate, *false* if deferred);

CondFileNames The name of the Algres procedure that contains the code that implements the evaluation of the trigger condition;

ActionFileName The name of the Algres procedure that contains the code that implements the trigger action;

Events The set of events that can activate the trigger;

EvFormulas The set of event formulas occurring in the trigger condition.

3.4 Organization of the Translation Process

The translation from Chimera to Algres operates in two major steps.

1. First the TRIG and TRANS compilers produce the intermediate code representation in IGLOO; compilers use VICHI-CL, a library of C++ programs which provide services for the intermediate storage of IGLOO representations into nonpersistent data structures. In addition, information about triggers is persistently stored in the Trigger Data Dictionary as described in the previous subsection.

2. The second step consists in generating Algres code from the IGLOO representation. The information driving the translation process is obtained by navigating the IGLOO graph and querying the Data Dictionary. Algres code produced is decomposed in procedures: each transaction line is encapsulated in one procedure; each trigger is implemented by two procedures, one for the condition, the other for the action.

Although the Year-1 compiler produces ALGRES definitions for both the horizontal and vertical alternative implementations of generalization hierarchies, in the Year-2 compiler we have chosen to use the vertical implementation. The motivation of this choice is that in Chimera we allow for the same real-world object to belong to more than one "most-specific" class (e.g., a person P that is at the same time an employee and a student); with this assumption, the horizontal solution becomes very hard to manage, due to the existence of replicated data (in the above example, the state of P is replicated in both the employee and student classes).

3.5 Computation of Event Instances

An *event table* is used to store event instances that have occurred during execution; event tables are updated at each new event. This forces us to generate Algres code for two purposes:

- Database-related Algres code which reads and updates the Algres Database in order to implement the Chimera primitives;
- Trigger-related Algres code which updates the *event table*, to keep track of database processing.

This happens because in our active database prototype we have no internal mechanism for detecting and automatically propagating events into event table; therefore we need to explicitly program such event manipulations. This is an area where there is a significant potential for optimization, however at the price of developing a mechanism which operates "from inside" the database kernel.

Updating event tables with generalization hierarchies is particularly critical, because events at given levels of the hierarchy may propagate at different levels. In particular:

- Creation or deletion actions on given levels of a generalization hierarchy need to be propagated appropriately; for instance, hierarchy $C1 < C2 < C3 < c4$, creation or deletion of object $o3$ in $C3$ propagates to $C2$ and $C1$. Note that, due to the assumption of multiple "most specialized classes" for the same object instance, deletion of an object might propagate also to "sibling classes" (e.g., if $C1$ and $C2$ inherit from C, when an object belonging both to $C1$ and $C2$ is deleted from $C1$ it must be also deleted from $C2$; the event deletion must be appropriately propagated).

- Attribute-specific event instances (queries or updates) which are recognized at a given level of the hierarchy propagate an event instance for each level where the property is inherited: thus, in a hierarchy $C1 < C2 < C3 < C4$, with an attribute A defined in $C2$ and inherited by $C3$ and $C4$, an update to object $o1$ belonging to class $C3$ is observed also for $o1$ as class instances of $C2$ (but not of $C1$); in addition, if $o1$ belongs to $C4$, then the update is also observed in class $C4$.

We have chosen of performing the analysis and generation of event instances by means of code that is generated at compile time (an alternative solution, probably less efficient, would be to assign event instance propagation to the run time).

To translate the *holds* and *occurred* predicates, we have also introduced another auxiliary ALGRES structure (called *predicate table*), which contains the *OID*s of the objects satisfying the predicate. The management of this table takes advantage of the ALGRES programs which determine event instances; indeed, these event instances are stored in an *event tree*, a fast-access data structure maintained by the run-time system, see Section 4. Thus, compile-time management of the predicate table would generate unnecessary overhead in this case; in order to improve efficiency, the management of this table is left under the control of the run-time system (while all other ALGRES tables are manipulated by effect of ALGRES programs which are generated by compilers).

The translation of the *old* predicate requires another auxiliary structure (the *Log*), which instead does not require a runtime support, but is entirely managed by the Algres code generated in this step. The *Log* structures have been designed but not yet implemented; they would anyway be produced only for triggers which use the *log* predicate, and restricted on the objects in the schema which are selected by that predicate.

3.6 Translation of Procedural Expressions

In the Year-2 compiler we consider only procedural expressions that use the *sequence connector*, omitting the *pipe connector*. Each transaction line is translated into an Algres procedure; procedures were specifically designed for their use in the context of IDEA and are described in Section 6. In this way, all the variables are defined locally to an Algres procedure and cannot interfere with other variables in the Algres program (similarly, the scope of a Chimera variable is limited to the transaction line in which it appears, and there is no need to pass information between different transaction lines). Further, it is possible to use standard names for all the auxiliary tables and the binding tables, which contain values that are passed between the primitives of the transaction line. The Algres code results clearer and more readable. All procedures have the following structure:

- Declaration of the structures used to keep the bindings
- Declaration of auxiliary local structures
- Binding computation

- Primitive execution

Actually, *Event Tables* are the only data structure which is shared between procedures; they can be considered as global to them.

The translation of a transaction line is decomposed into two different levels.

- The higher level is performed by the *Main Translator (MaTra)* which analyzes all the primitives of the transaction line and variables passed from one primitive to the other.
- The lower level translators generate code for each individual Chimera primitive and have a very limited view of the context of the translation.

Such a modularization implifies the translation process: multiple small translators, with very similar structures, have been built, one for every different type of primitive.

According to the semantics of procedural expressions in Chimera, a variable is associated to a set of bindings when it is introduced by a *select* or a *create* clause, and then these bindings cannot change; of course, they are passed to all subsequent primitives which use that variable as an input parameter. An ALGRES table (called *binding table*) is associated to every primitive producing output variables.

3.6.1 Main Translator

The *Main Translator (MaTra)* performs the following tasks:

- Defining the ALGRES structures used to store the bindings for input and output variables (called the *input and output binding table*; all bindings for input variables are thus stored in a single ALGRES table).
- Computing the cartesian products for of bindings of input variables and store them in the input table (the cartesian product here does not represent an overhead because the semantics of procedural expressions of Chimera is exactly to consider all possible combinations of bindings for the input variables.
- Linking (via ALGRES links) the input variables to the referenced ALGRES database tables. Links have been added to Algres in order to facilitate the navigation based on OIDs (which is very frequent due to isa hierarchies and object-valued attributes); for their description, see Section 6.

Two auxiliary structures are present in every procedure, called *Evcompile* and *Evrun*; these contain records inserted after the execution of every primitive that generates an event. *Evcompile* contains the information known at compile-time, i.e. the type of the primitive and the names of the attribute and of the class on which the event occurs, while *Evrun* contains the information that can only be acquired at run-time, like the *OID*s of the objects that have been modified or accessed by the primitive.

3.6.2 Individual Translators of Chimera Primitives

Chimera's primitives can be divided in two distinct categories:

- Primitives that produce bindings (*select*, *create*).
- Primitives that only consume bindings (*modify*, *display*, *delete*, *generalize*, *specialize*).

Each of the above primitives is assocated to a different translator.

A simple syntactic analysis distinguishes input and output variables: for instance, for a select primitive, output variables appear at the left of the `where` clause, the remaining ones are eithr input variables or locally defined ones; locally defined variables are recognized because they are introduced by a type or class formula. In the following, we consider the translation of the *select* primitive in the more general setting of declarative expressions.

3.7 Translation of Declarative Expressions

Declarative expressions are translated in the context of the primitives *select* and *display*, and in the translation of *conditions* of triggers; aggregate expressions may recursively involve declarative expressions. They are subject not only to translation, but also to optimization; indeed, due to the complexity of the declarative expressions in Chimera, such optimization goes beyond conjunctive query optimization and encompasses path expressions, aggregates, set and list management. Note that ALGRES, similar to FLORA, is a good vehicle for expressing the *result* of optimizations, as it is a low-level procedural language, although algrebraic in nature, and therefore can be used to describe an execution strategy with enough details. So far, however, we have not spent much efforts in the optimization of declarative expressions; in particular, we do not envision to have a cost-based optimization (since we do not expect to collect any statistics of the underlying database tables); therefore, optimizations are limited to few obvious choices, such as executing equijoins before general theta-joins and anticipating the evaluation of selections.

The general approach to translation is illustrated by the following sequence of steps, which progressively builds a *result table* as follows:

1. Assume initially the result table equal to the *input binding table* (possibly void); when trigger conditions are processed, the input binding table is given by the cartesian product of instances bound to the *occurred* and *holds* predicates.
2. Repeat the following two steps whenever applicable:

- Join the input binding table to Algres tables corresponding to the local object variables which are locally defined; joins are based on comparison conditions established between input variables and local variables.
- Extend the result with *tuple extension* operations whenever a local object variable introduces an object which is *reachable*, by means of object-based attributes, from objects which have been joined already.

3. If the above steps complete and there is some local object variable which has not been linked to the input binding table, then apply the cartesian product between the table representing one of these varianbles and the current result table, and assign the result to the current result table; then repeat steps 1 and 2 until *all* local variables are joined together.

4. If an aggregate expression has a where clause, then recursively call at this point this same translation algorithm; the current result table constitutes the context in which the nested declarative expression has to be evaluated. Eventually, the aggregate function is computed on the result produced by the recursive call and linked to the result of the calling context (by means of the ALGRES tuple extension).

5. Apply next all aggregate queries that do not introduce the need for a new declarative expression evaluation (by means of the ALGRES aggregate operation and tuple extension).

6. Apply next all selections involving arbitrary predicates.

7. Apply next the projection on the target variable.

This procedure is post-optimized by *pushing* selection conditions deep in the resulting operator tree whenever possible (i.e., by applying the selection at the earliest possible time) and by dropping parts of the result relations whenever they are no longer used (by applying additional projections).

3.8 Examples of Translation

In this subsection we describe the result of the application of the translators to the example used in the description of the IGLOO language. This example contains two *select* actions followed by a *modify*.

```
select(X where project(X),
                {[``Paolo'',''Rossi''],[``Marco'',''Bianchi'']} in X.members),

select(Y where project(Y), Y=X, integer(I),
                 I=sum(Z.resourceavailability  where DBplanningperiod(Z),
                                                      Z.startdate='1/1/92',
                                                      Z.enddate='1/1/93'),
```

```
                integer(N), N=card(W where project(W), W=X),
                Y.resourceneeded>I/N),

modify(project.maxresourceallocable, Y, maxresourceallocable*1.1);
```

The following Algres code translates the *select* primitive:

```
DEF action1:
<
X:integer
>

DEF action2:
<
Y:integer
>
```

First are introduced the *binding tables* `action1` and `action2`, which will contain the set of objects that correspond to variables `X` and `Y`.

```
LINK action1Xproject [action1.X->project.Self ]
LINK action2Yproject [action2.Y->project.Self ]
LINK projectmembersperson [project.members.Item->person.Self ]
LINK projectYaction2 [ project.Self -> action1.Y ]
```

Then, Algres links are introduced; they constitute local auxiliary structures that allow to reach all the object attributes from the *OID*s contained in `X` and `Y`.

```
action1<-
PJ[X]
SL [CONTEQ({("Paolo","Rossi")("Marco","Bianchi")}; Xmembers.members )]
TE [ Xmembers.members.name:=projectmembersperson->name]
TE [ Xmembers.members.sname:=projectmembersperson->sname]
RN [Self->X, members->Xmembers,
             resourceavailability->Xresourceavailability ]project
```

The first action first renames the attributes of class `project` (using the rename operator `RN`); then the name and surname of the project members are assigned to the new attributes `sname` and `name` (using the tuple extension operator `TE`); of all the projects are selected only those that have as members both `Paolo Rossi` and `Marco Bianchi` (using the selection operator `SL` and the set comparison predicate `CONTEQ`). The action keeps only the *OID*s, so a projection on the `Y` attribute is executed (using operator `PJ`).

```
action2<-
PJ [ Y ]
SL [ Yresourceavailability > I/N ]
TE [ N := CARD(action1) ]
TE [ I := AF[+/resourceavailability ]
     SL[ startdate="1/1/92" ]
     SL[ enddate="1/1/93"]DBplanningperiod ]
JN [ Y=X ]
RN [Self->Y,members->Ymembers, resourceavailability->Yresourceavailability]
project
action1
```

The second action uses the result of the first action and the information contained in the `project` table. It first renames the attributes, then it selects only the tuple projects that have already been selected by the first action, with a join between the two tables (using the join operator `JN`). An attribute `I` is added to the schema which contains the sum of the resource availabilities. Attribute `N` is added, to contain the number of tuples of `action1`. Finally, a projection is executed on the result.

```
project<-
TU [ maxresourceallocable:=maxresourceallocable*1.1 ]
SL [ projectYaction2 -> Y = Self ] project

evcomp <- <("modify", "project", "maxresourceallocable")>
evrun <- SL [ projectYaction2 -> Y = Self ] project

IF (NOT EQUAL( evrun, <> )
THEN
[
 CALL EVENTS ( events, evrun, "modify", "project", "maxresourceallocable")
]
```

These lines translate the *modify* action, increasing the value of attribute `maxresourceallocalble` by 10 % for the projects that had been previously selected. After having executed the update, the program updates the `evcomp` and evrun tables. If the set of events is not null, a procedure `EVENTS` is called to update the *occurred events* and *Log* tables.

4 RUN TIME SUPPORT

Contributed by: S. Castangia, G. Guerrini, P.P. Merli, L. Colella

The run time support provides the following functionalities:

1. It reacts to the transactional commands, basically to administrate the execution environment that has a transaction as lifetime, i.e. structures and control information that are initialized at begin transaction and should be emptied at commit.
2. It calls the ALGRES machine for the execution of compiled code, both in normal and in reactive processing.
3. It manages reactive processing fully implementing the semantics of Chimera active rules; in this case run time activation occurs automatically at specific points (the end of each transaction line, or at each save point and commit command).

All the features of a run time support for active rule processing, as described in [18], have been implemented with the unique exception of the log mechanism. Thus, *old* predicates are not handled in the current Chimera testbed.

A timestamp mechanism is used to coordinate the various components of active rule processing, i.e. the event base and the information about rules. Our (logical) timestamp is an integer that is set to zero at the transaction beginning and is increased by one after each occurred event (update operations or queries). Each event is recorded with a timestamp, which indicates when the event happens. Each rule is recorded with the timestamp of its last inflection point (last completion of rule evaluation).

4.1 Control Flow of the Run Time System

To describe the control flow of the system, we consider a generic execution. The run time system is organized in modules; each module is implemented by a class in C++.

The activation of the RTS process is caused by the UFI, which asks the run time system to execute single Chimera transaction lines after their translation into ALGRES expressions.

Transactional Commands are directly interpreted by the run time system through the *Transactional Command* module, which, on reception of the command `openDB(nome_db)` simply opens the ALGRES database. Another consequence of this command is the fetching of the rule data dictionary from the ALGRES database; indeed the run time system maintains a private main memory version of these information about rules throughout the Chimera session. The run time system maintains also an *event tree*, that is, a tree of event types, which is used for an efficient determination of the triggered rules from the set of occurred events. This tree is initialized upon `openDB` by inserting the event types triggering some rules, while it is updated as soon as events of new event types occur during the session.

The operation that usually should follow an `openDB` command is the `begin transaction` statement. From the run time system viewpoint this statement causes the initialization of some internal structures, namely the Occurred Events and Triggered Rules tables.

Now, a transaction line is expected. The external interface send it to compilers and once the transaction line compilation is completed, the interface calls in turn the run time system for executing the compiled transaction line, whose code has been stored in a determined file. At this point, the RTS module calls the *Transaction Line* module, which sends the transaction line to be executed to the underlying ALGRES machine. The ALGRES machine, after having completed the transaction line execution, returns to the run time system the events that have occurred as a consequence of the transaction line execution.

Events are handled as follows. The *Transaction Line* module takes care of signaling occurred events received from the ALGRES machine to the *Occurrent Event Log* module, which performs the task of storing occurred events. Events must be correctly inserted in the table. Thus, for each individual event, the corresponding event type must be searched for in the event tree. After this search we are able to know the timestamp of the last occurrence of an event with the considered event type, stored in the event tree, and thus to keep the events ordered; all rules triggered by the event are also extracted. The information on the last occurrence of an event with the specified event type is updated to take into account also the last occurred event.

If the search for the event type in the tree fails, then no events with the considered event type have occurred during the session. Since the tree has been initialized with all the event types triggering rules, we know that such an event does not trigger any rule. However, we currently stored all events because they might be used in the evaluation of event formulas in some rule condition. This happens because Chimera permits the use of event formulas on any arbitrary event, without restricting event formulas to triggering events; a possible optimization consists in collecting the information about predicates used in event formulas and then logging only these events into the event tree.

After the execution of each transaction line, and the storing of the events it caused, active rule processing is activated by invoking the *Rule Table* module. If the rule table contains some triggered immediate rules, then the rule among them with higher priority is extracted to be executed; in terms of control flow among internal modules of the run time system this operation causes the control to be ceased to *Rule Execution* module. The execution of a rule consists first of all in the evaluation of the event formulas possibly appearing in its condition part. This task is performed by the *Event Formula* module. After that, the *Rule Execution* module executes the rule condition and (possibly) its action: a reference to the rule compiled code is contained in the rule data dictionary. This execution may be performed either atomically or with an user interaction, depending on the chosen option (see below).

Generally speaking, the execution of a rule action causes operations on the database, and thus the occurrence of new events. The Rule Execution module invokes the *Occurred Event Log* module for storing new events. Note that those new events may trigger new rules, and thus all triggered rules are executed till a quiescent state is reached. Then, active rule processing stops, and the following transaction line is processed.

A `savepoint` statement may also be encountered in the execution of a transaction. This statement causes the activation of active rule processing considering both immediate and deferred rules. The same behavior with respect to rule execution is taken in case of a `commit` statement is encountered. However, a `commit` statement also signals the end of the transaction, thus data structures whose lifetime is a transaction, such as the Occurred Events table, are emptied and the ALGRES commit is executed. The ALGRES commit causes the database updates in main memory to be permanently stored in the database.

A Chimera transaction may be terminated by a `rollback` statement, instead of by the commit statement. A `rollback` statement causes the run time system simply to empty the data structures and to invalid the transaction (no ALGRES commit is executed, so the secondary memory database is that corresponding to transaction start).

In the following we describe separately and at a greater level of detail each of the tasks performed by the run-time system.

4.2 Execution of Executable Entities

One of the key task of the Run Time System is the support for the execution of the above mentioned executable entities. Executable entities may be partitioned in two subclasses:

- Executable entities whose execution is explicitly requested by a system module external to the run time support
- Executable entities whose execution is not requested from outside the run time system, rather the execution is requested by the run time system itself.

To the first group of executable entities belong transactional commands and transaction lines. The second group of executable entities consists only of rules.

Transactional Command Execution
Transactional commands are not compiled in Chimera. Thus, when the external interface receives a transactional command, it is passed to the compilers only to check its syntactic correctness, then, if correct, it is passed to the run time system, which must execute them directly. In the following, for each *Transactional Command*, we specify the operations to be done by the Run Time System.

We remark that the lifetime of the Run Time System process is a whole Chimera Data Manipulation (DM) session; by contrast, the Run Time System process is not active in case of Data Definition (DD).

openDB The first *Transactional Command* we consider is `openDB(nome_db)`, which requests to open the database whose name is `nome_DB`, in such a way that the user may work on it during the session; the Run Time System simply opens the Algres database and initializes its data structures. More specifically, upon an `openDB` command

- An event tree, containing only event types that trigger some rule, is built
- The rule data dictionary is fetched from the ALGRES permanent structures, and the rule table is initialized with the available rule information
- The occurred events table is initialized (obviously, in this phase the table is empty, since no event is occurred).

begin transaction The `begin transaction` statement opens a transaction and requires that some Run Time System data structures are reinitialized. In particular, those structure whose lifetime is a transaction, must be emptied. More specifically, the Occurred Events table must be empty and the rule table must have an empty conflict set. As far as the event tree is concerned, it might contain only those event types that trigger some rule (as at openDB). Alternatively, the event type of events occurred in the previous transactions of the same session might be not deleted from the event tree.

rollback The `rollback` command ends a transaction and states that the transaction does not have success. In this case, the Run Time System must invalidate the transaction, that is the updates performed by the transaction on the database must not be made persistent. In case of a transaction ended by a `rollback` command, the active rule processing is not performed at all, since useless.

commit The `commit` command ends a transaction and states that the transaction correctly terminates. Thus, the Run Time System starts active rule processing considering also deferred rules. When active rule processing reaches a quiescent status, the updates performed by the transaction are permanently stored in the database. We remark that the action part of a rule may contain a rollback command, which invalidates the top level commit. Thus, the actual commit can be executed only when active rule processing is ended.

savepoint The last transactional command we consider is `savepoint`, which may appear as internal command in a transaction. The semantics of such a command is to cause active rule processing both on immediate and on deferred rules.

Transaction Line Execution

The second kind of executable entity belonging to the first class (execution is requested from outside RTS) is the transaction line. Each submitted transaction line is passed from the graphical interface to compilers to be compiled. Its compiled code is stored in a determined file. After the compilation, compilers return control to the interface, which, in turns, activate the Run Time System, which gets the compiled transaction line to be executed by the underlying ALGRES machine.

Rule Execution
The only executable entity belonging to the other class is the rule. Indeed, a rule is activate only if, during the execution of transaction line, an event is occurred whose event type appears in the event part of the rules. When a rule is triggered, it is inserted in the *conflict set* to be then executed as soon as its priority becomes the greatest.

4.3 Event Handling

Events in Chimera are generated by all ALGRES operations executed as a consequence of Chimera operations contained in transaction lines and in rule action part. We remark that queries (select) and the condition part of a rule do not generate any event.

When the ALGRES machine executes the entities which generate events, it reports the occurred events to the Run Time System. Each occurred event is equipped with the timestamp of its occurrence, that denotes the moment in time when the event has occurred. Moreover, each occurred event is related to all the objects in the database affected by the operation, thought as set oriented. Thus, the Run Time System sees an occurred event as a triple

$$\langle Event\ Type,\ Timestamp,\ Set\ of\ Oids \rangle$$

where *Set of Oids* represents (through their identifiers) the objects affected by the operation.

Occurred events are key entities in the Run Time System, indeed they allow to handle rule activations and to evaluate event formulas. Intuitively,an event activates a rule when its type corresponds to a type among those specified in the event part of the rule.

As soon as an occurred events is received from the ALGRES machine, the event is stored in the Occurred Events table, through a search in the event tree. The event tree is the structure that allows a fast access both to the Occurred Events table and to the rule table. Indeed, through the entry in the tree for a given event type we know which rules must be activated on the event occurrence. Thus, the system accesses the rule table and activates those rules (namely, it inserts them in the conflict set). If a rule is already triggered (it has been activated by a previously occurred event, and it has not yet been executed) then, no action is done for this rule.

After having determined the set of rules activated by an occurred events, events are stored because they might be useful for the evaluation of an event formula. To properly evaluate an event formula we must return a set of OIDs, given a specified event type. In case of simple event formulas (*occurred* predicate) the set simply includes the identifiers of objects on which an event of the specified type has occurred. In case of event formulas with net effect (*holds* predicate) the set is computed by difference between OID sets (the set of OIDs for which the event has occurred minus the set of OIDs for which a "complementary" event has happened).

The result of an event formula evaluation is the set of the OIDs for which the predicate is true. The returned OIDs are a subset of those selected from the Occurred Event table.

4.4 Active Rule Processing

The support for Active Rule Processing (ARP) is the Run Time System basic functionality; in the following we briefly show how active rules are handled.

As seen, the occurring of an event causes the activation of all rules which contains in their event part the event type of the occurred event (possibly, some of these rules are already triggered). Thus, the conflict set contains a number of triggered rules, and the Run Time System task is to choose the next rule to be executed. This choice is influenced by different conditions:

- *Processing mode:* if we consider reactive processing activated at the end of a transaction line, and not by a transactional command like `commit` or `savepoint`, then only an immediate rule may be chosen; by contrast, when the reactive processing is activated by `commit` or `savepoint` we may choose either an immediate or a deferred rule
- *Priority:* we must choose the rule in the conflict set with highest priority; in case of equal highest priorities, a rule is chosen among those with highest priority non-deterministically.

After the rule selection, event formulas possibly appearing in the rule condition are evaluated. If this evaluation returns some bindings, then the condition part of the rule is executed. By contrast, if the evaluation of the event formula returns an empty set of bindings, the condition is not executed at all because we know in advance that it is false, because it is a conjunction with a false argument.

A timestamp is related to each rule. This timestamp records when the rule has been executed for the last time. The timestamp is relevant also for event formula evaluation. The timestamp of a rule is updated after the rule execution setting it to the value of the last occurred event timestamp, excluding those events generated by the action part of the rule itself.

The ALGRES code of the condition part is executed. After the ALGRES machine as evaluated the rule condition, it returns the result of the evaluation, through a set of bindings, to the Run Time System. If the condition part has been evaluated to true (a not empty set of bindings has been returned) then the action part is executed. By contrast, if the condition part has been evaluated to false, then we turn to consider another rule. To execute the action part of a rule, the related compiled code must be passed to the ALGRES machine, altogether with the set of bindings returned by the condition evaluation. Those bindings are stored in a data structure global to the Run Time System and to the ALGRES machine. An analogous global structure is used to pass the set of oids returned by the evaluation of an event formula (performed by the run time system) to the ALGRES machine that must evaluate the other components of rule condition starting only from the retrieved OIDs.

The execution of the rule action generates new events, that are returned to the Run Time System. Thus, new rules may be triggered and the conflict set must be updated. The reactive processing continues, selecting the new rule to be executed. Reactive processing stops when an empty conflict set is got.

4.5 Event Formula Evaluation

Event formulas may be built on two predicates: *occurred* and *holds*. These predicates have as argument a set of event types on which the predicate must be evaluated. The evaluaton of event formulas requires to know the occurred events stored in the corresponding table.

When a triggered rule is selected for execution, if its condition contains event formulas, these formulas must be evaluated before getting the rule condition executed by the ALGRES machine. Such predicates are evaluated on a set of occurred events. If a formula has as input parameter a set of event types ET, the set of OIDs is computed as the union of the sets returned by the evaluation of the predicate with a single event type in the set ET. In the following we briefly explain how these predicates are evaluated on a single event type.

Occurred predicate

Occurred events of the specified event type are selected. Moreover only events whose timestamp belongs to a specified interval are considered. The endpoints of the interval are the timestamp of the last occurrence of the event type and the rule timestamp, i.e. the timestamp of the last rule execution. In this way we select all the occurred events with the specified event type, not yet considered by the rule. The set which is returned as result is the union of the OID sets related to events selected according to the above criteria.

Holds predicate

The evaluation of the *holds* predicate requires to select the events of the specified event type that have occurred, according to the criteria explained above. The timestamp interval considered is the same as for the evaluation of the *occurred* predicate. However, we must also compute the events of "complementary" event type that may have occurred, by performing a difference of OID sets. The event type involved in the nettification process, in addition to the one specified as input, depends on the event type in the formula (see the consolidated specification of Chimera, [1]).

We finally note that the rule priorities that the Run Time System gets, together with all other information about rules, when it fetches the rule data dictionary, are static priorities. These priorities are computed starting from the graph that represents the rule partial ordering derived from `before/after` clauses in rule definitions; priorities are translated into a total order (integer numbers) which is nondeterministically chosen by the TRIG compiler and is not changed by the RTS. The UFI may require to the Run Time System to recompute rule priorities starting from the partial ordering graph, so as to enable the user to indicate which of the rules with the same priority should be executed first.

5 USER-FRIENDLY INTERFACE

contributed by: P. Fraternali, P. Rossetti, D. Roda

We recall that a Chimera database can be accessed in one of two ways:

- An application program written in an external language can embed a set of Chimera database query and/or manipulation operations, that are submitted to the Run Time System (RTS) via an Application Programming Interface (API).
- The end-user (either the application user or the database administrator) can directly interact with the database management system, via a User Friendly Interface, that permits both the extension of the database schema and the manipulation and inspection of the database instance.

In the current implementation of the ALGRES Year-2 testbed, we have developed a simple API that enables the ad-hoc linking to the TMS application of TXT. All our efforts have instead been concentrated on interactive use of the testbed through the UFI. In this Section, we present the features of the implemented prototype of the UFI.

5.1 Global Architecture of the UFI

The general principle adopted in the design of the UFI is the separation between the interface to extend the schema of the database, and the interface to manipulate and query the instance of a database.

Hence, the UFI is structured into two main modules:

- UFI-DD supports all the activities related to data definition, namely the browsing of the schema of an existing database and its extension. An existing schema can be extendend by adding new classes/value types (both their signature and implementation) and new triggers. When a transaction comprising a set of data definitions, submitted to the UFI-DD, is successfully committed, the data dictionary of the database is permanently updated to reflect the changes.
- UFI-DM support all the activities connected to the manipulation of data. In Chimera data updates are of two types: transaction and rule updates. The former

are caused by a transaction supplied by the user, whereas the latter are due to the reaction of the active database to the user's updates. Hence, the UFI-DM consists mainly of two components: a Transaction Manager, whereby the user can edit, compile and execute a transaction, and a Rule Monitor, which helps the user in tracing the execution of active rules.

5.1.1 Interface to Other Subsystems

The UFI communicates with all the other components of the Chimera architecture.

Both UFI-DD and UFI-DM dialogue with the set of federated compilers that support Chimera rich conceptual language. The communication is bidirectional: a (set of) self-contained syntactic unit(s) (e.g., a class definition, a trigger or a transaction line) is sent from the UFI to the appropriate compiler, which compiles it and sends back a response message to the UFI, which in turn presents the outcome of the compilation to the user.

UFI-DM has also a tight link to the Run Time System: after the successful compilation of a transaction unit, a command in the UFI can be used to submit the compiled code to the RTS for execution. In turn, the RTS sends back to the UFI information on the execution of the transaction unit and possibly of the triggers activated by it.

If the rule monitoring function is activated by the UFI, then an even tighter connection is established between UFI-DM and the RTS. In this case, the execution of the triggers is driven by an interactive interface, which allows the user to halt the execution of triggers, inspect the past trace and the actual state of execution, and even intervene on the conflict resolution strategy, to direct rule processing.

5.1.2 Implementation Standards

The UFI is implemented in C^{++}, for the control part, and in *Motif*, for the presentation part.

It consists of two distict processes (one associated to UFI-DD, the other to UFI-DM), which, when needed, communicate through Unix pipes with the other processes constituting the Chimera system (namely the federated compilers and the RTS). Each process has its own graphic interface, implemented as a *Motif* main window.

A third interface serves the Rule Monitor. This interface has been implemented directly on top of the RTS, due to the performance problems that would arise from the inter-process communication overhead, if the Rule Monitor were physically part of the UFI process.

Figure 15: The main window of UFI-DD

5.2 Data Definition Interface

The UFI for data definition (UFI-DD) permits to selectively open one of the defined databases, browse its schema, and possibly extend it with new definitions.

Schema extension is achieved by compiling a set of Chimera declarations, comprising any of the features supported by the Year2 prototype, namely:

- The signature of a new class;
- The implementation of an existing unimplemented class (at the moment, it is possible to specify only the trigger components of a class implementation);
- the signature of a new value class.

A typical working session with the UFI-DD consists in specifying the target database, opening it, defining a schema updating transaction comprising a set of Chimera declarations (this can be done either by loading an externally defined text file or by using the internal editor to actually write the transaction), submitting it to the compiler, and receiving back the outcome of the compilation.

In case the compiler detects no syntactic or semantic errors, the transaction is committed and the persistent data dictionary of the target database is updated to reflect the extensions.

If only syntactic errors are detected, the compilation process is completed, the transaction is rolled-back, and the user is presented with the errors encountered during the compilation.

If an unrecoverable semantic error occurs, the compilation is stopped and the error presented to the user, which must correct it and re-submit the transaction.

Fig. 15 shows the interface of UFI-DD.

5.3 Rule and Transaction Interface

The UFI for data manipulation (UFI-DM) permits to selectively open one of the defined databases, compile and execute a transaction formed by data queries and manipulating operations, and possibly control the execution of triggers activated by the transaction.

5.3.1 Global Architecture of the Rule and Transaction Interface

Due to the complexity of the interaction with a sophisticated active database, the UFI-DM has been given a modular structure, described in Fig. 16.

Figure 16: The architecture of UFI-DM

The interface is organized into two major components, each one associated to a separate window. The first component is the Transaction Manager, which acts as a control panel for all the activities related to the definition, compilation and execution of a database updating transaction. The second component is the Rule Monitor, which is specifically devoted to the task of tracing the execution of active rules. The Rule Monitor is activated from the Transaction Manager to interactively debug the execution of active rules and its interface is displayed beside the Transaction Manager main window; when the normal execution mode of rules is resumed, the Rule Monitor is automatically switched off.

Both the Transaction Manager and the Rule Monitor are described in detail in the following Sections.

5.3.2 The Transaction Manager

The Transaction Manager offers the same functionalities, for database updating transactions, as the UFI-DD for schema updating transactions: it provides an interface by which the user can follow all the activities ranging from the definition of a database updating transaction to its submission to the RTS.

Accordingly, the interface of the Transaction Manager is organized into two major areas (see Fig. 17), called the Editing Area and the Compilation and Execution Area.

Figure 17: The interface of the Transaction Manager

The Editing Area is the part of the Transaction Manager interface where the user can write a database transaction and/or load and possibly modify an existing stored transaction, prior to invoking the compiler. The text of the transaction to be compiled is shown in an updatable text window within the Editing Area and a simplified text editor is provided to facilitate the input and modification of transactions.

After a transaction has been input in the editing area, its compilation and execution can proceed in two ways: at a transaction unit by transaction unit pace or globally for the whole transaction.

In the former mode, the transaction is segmented into transaction units (each but the last one ended by a ";") and a command exists to submit each unit individually to the compiler and to execute a (set of) compiled transaction unit(s).

In the latter mode, all transaction units from a given point in the transaction to the end can be compiled and executed in a single step.

The layout of the Compilation and Execution Area includes the following elements:

- A read only text window, which records the transaction units that have been already successfully compiled and executed;
- A compilation command area, that contains two click buttons to activate the compilation in the one step or unit by unit mode;
- A compilation message area, i.e., a pop-up dialog box that displays the return messages from the compiler;
- An execution menu, that contains items to submit compiled transaction units to the run time system and to activate the Rule Monitor; this menu also contains a command to display the execution trace;
- An execution context area, which graphically shows the current context of execution (i.e., if a user transaction unit or a rule is being executed) and records the number of executed rules.

The execution trace command in the execution menu displays a pop-up text window showing information about the rules already executed; in particular, for each step of the reactive processing, the trace contains the set of triggered rules (called *conflict set*) and the rule that was chosen for execution.

5.3.3 The Rule Monitor

The Rule Monitor is an interactive interface to inspect and direct rule processing.

After a transaction unit has been compiled and submitted to the RTS, reactive processing can be activated in two ways:

- in the *continuous* mode, the transaction unit is executed and reactive processing awakened without intervention of the user; when a quiescent state is reached, execution control is passed back from the RTS to the Transaction Manager, for the compilation and execution of the next transaction unit.
- in the *debugging* mode, the transaction unit is executed and the control is transferred to the Rule Monitor, whereby the user can perform various actions to obtain information about rule processing and to command the execution of rules.

The continuous mode is entered by selecting the *Run* command in the rule execution menu of the Transaction Manager, which has the effect of sending to the RTS all the compiled, but not yet executed, transaction units. The RTS executes each transaction unit, followed by rule processing, without activating the Rule Monitor.

The debugging mode can be entered at any time during continuous rule processing, by using the *Debug* command in the rule execution menu of the Transaction Manager. When this command is executed, an asyncronous message is sent from the Transaction Manager to the RTS, which completes the execution of the current rule, exits the continuous rule processing mode, activates the Rule Monitor interface, and starts executing in the step by step mode.

Interactive execution of rules is available at two levels of granularity:

- At the *rule step* level, reactive processing is halted and the situation presented to the user after each rule execution; there is no possibility of interrupting the run-time system during the computation of triggered rules and the evaluation of rules' conditions;
- At the *intra-rule step* level, reactive processing is halted in correspondence of the three fundamental moments of rule execution: triggering, condition evaluation and action execution. The user can obtain information on the state of rule processing and influence (in a limited way) the behavior of the rule executor.

The Rule Monitor offers functionalities common to both the rule and intra-rule debugging mode, plus specific functions available only in the intra-rule debugging mode.

The following functionalities are available independently of the debugging mode:

- *Inspection of the Conflict Set*: after the execution of a transaction unit or of a rule, the RTS computes the set of triggered rules, called *Conflict Set*. The names of the rules in the conflict set are displayed in a read-only scroll list, in order of priority. Rules with the same *static priority* (i.e., the priority deriving from the declaration of triggers) are ordered according to their *dynamic priority*, which is a total order among different triggered rules with equal static priority, computed by the RTS. Rules in the list can be individually selected, by clicking on them with the mouse left button.
- *Inspection of the Deferred Conflict Set*: normally the conflict set contains only immediate rules, since deferred triggers are activated only at transaction commit time. At any point in rule execution, it is possible to ask the RTS for the virtual conflict set that would be computed, if also deferred rules were taken into account.
- *Information on Rules*: when a rule in the conflict set is selected, the Rule Monitor displays, in a read only text area, information about the rule, including: the activation and consuming mode, the static and dynamic priority.
- *Conflict Resolution Strategy Selection*: since the static priority of triggers defines a partial order, it is possible that multiple rules with the same static priority are triggered simultaneously. In this case, the RTS selects the rule to be executed based on dynamic priorities. Dynamic priorities are assigned to rules by the RTS in two ways:
  - *Deterministically*: a set of rules is always ordered in the same way, irrespective of the execution context.

  - *Non-Deterministically*: at each computation of the conflict set, rules with equal static priority are ordered in a random way, so that there is no guarantee that the same rule is selected in two different computations of the conflict set, even if all the circumstances are the same.

  The Rule Monitor permits to switch from the deterministic to the non-deterministic conflict resolution strategy, and vice versa.

- *Rule Activation/Deactivation*: triggers can be explicitly deactivated, so that the RTS disregards them during reactive processing. A deactivated trigger can be made active, even at intermediate states of rule processing.

- *Information on Occurred Events*: all events since the beginning of the transaction are displayed in a read-only scroll list; each event is described by its event type, the list of the *OIDs* of the objects affected by the event, and the indication of the database state in which the event has occurred. For identifying the various states of the database, all intermediate states since the beginning of the transaction are numbred progressively. Events are listed in order of occurrence.

- *Rule Source Code*: it is possible to display on demand the source code of any rule defined in the rule dictionary. The source code is presented in a read-only pop-up text window.

When the debugging mode is intra-rule step, the following additional options become available:

- *Processing Status*: a graphic icon is used to show the current point of the computation of the RTS: this can be either the computation of triggered rules, or the evaluation of rule conditions, or the execution of the selected rule's action.

- *Detection of First Executable Rule*: when the intra-rule step point is before the evaluation of the condition of triggered rules, it is possible to perform the evaluation interactively. A command is available to find the rule with highest priority, among those that have not been considered yet, whose condition is satisfied in the current database state.

- *Modification of Dynamic Priorities*: when the intra-rule step point is before the evaluation of the condition of triggered rules, it is possible to act on the dynamic priority of triggered rules with equal static priority. In this way, it is possible to force the RTS to select a rule different from the one that would be chosen if the default conflict resolution strategy were adopted.

- *Information on Condition Bindings*: when the intra-rule step point is before the execution of the selected rule, it is possible to inspect the bindings computed by the evaluation of the rule's condition; these are presented through AlGraph, the graphic interface of Algres, the underlying database of Chimera.

Fig 18 shows the layout of the Rule Monitor.

Figure 18: The interface of the Rule Monitor

5.4 Future Development

Several additional capabilities have been identified in the requirement collection phase, and could be considered for inclusion in the second version of the UFI. Among these, the most important are:

- The capability to display in a compact way, possibly through graphic icons, the whole execution trace. Icons forming the trace should be active, so that the user can obtain various kind of information, simply by clicking on the appropriate icon (e.g., the icon representing a rule in the conflict set may have a command attached, to retrieve relevant pieces of information about the rule).
- A support to the what-if execution of rules, to allow the actual debugging of preliminary versions of rules, without hampering the content of the database.
- The integration of the UFI and the Year3 CASE tools for the design of rules.

6 ALGRES++

Contributed by: G. Lamperti

This section is intended to informally describe the novel features introduced in ALGRES to cope with the efficient, modular, and extensible implementation of CHIMERA language. The new features of ALGRES are:

- Blocks and control structures,
- Conditional expression,
- Subprograms,
- Recursion,
- Links.

6.1 Blocks and control structures

Control structures are introduced in ALGRES not to extend the computational power of the language, since this power is complete thanks to the closure operator; rather they provide ALGRES programmers with a more rich and flexible paradigm to specify applications. Control structures are to be considered special kinds of statements involving other statements.

Two types of control structures are defined: conditional statement and loop statement. The former represents a generalization of the common *if-then-else* construct; the latter corresponds to the while loop of traditional general-purpose programming languages, with the added possibility that the involved predicates and statements can be of arbitrary complexity.

Thus, the programmer is not constrained to use the closure operator when computing recursive algorithms. Furthermore, he or she is not forced to implement the procedural part of applications using ALGRES embedded in general purpose programming languages, such as the ALICE interface.

The syntax of control structures is the following:

$ControlStatement \rightarrow ConditionalStatement \mid LoopStatement$
$ConditionalStatement \rightarrow IF\ predicate\ THEN\ block$
$(ELSIF\ predicate\ THEN\ block)^{*}\ (\epsilon \mid OTHERWISE\ block)$
$LoopStatement \rightarrow WHILE\ predicate\ DO\ block$
$block \rightarrow BlockStatement \mid [\ BlockStatement^{+}\ ]$
$BlockStatement \rightarrow TypeDef \mid AssignStatement \mid IoStatement \mid$
$ControlStatement \mid ProcedureCall \mid LinkStatement$

Control structures are a special kind of statement recursively embodying blocks of other statements. A block is conceived similarly to ordinary blocks of statements of traditional programming languages. Thus a block in the ALGRES language is a sequence of ALGRES statements, i.e. a piece of ALGRES program.

Actually the set of block statements does not coincide with that of ALGRES statements. In particular, persistency statements and subprograms declarations (subfunctions and procedures) are not allowed in blocks. The reader can note that local static objects are allowed by the object definition statement. This entails a discipline in objects visibility, i.e. the rules determining the scope of local objects. These rules are very closed to those adopted in structured programming languages like Pascal or C.

6.1.1 Conditional Statement

The conditional statement is syntactically similar to the conditional expression. The main difference consists of the fact that the former is a statement while the latter is an expression, i.e. something returning a value. Actually, the conditional expression associates to each condition a candidate expression, while the conditional statement associates a block of statements. Thus a conditional statement cannot be the operand of any operator and, conversely, a conditional expression cannot be used as a statement but has to be included as value in a proper expression.

The following example shows a simple conditional statement (instanciations are omitted):

```
DEF R: { A: integer, B: string }
DEF cond: bool
IF cond THEN
  DISPLAY[] R
OTHERWISE
  R <- SL [ A > 0 ] R
```

Note that blocks are represented as single statements, respectively as a display and an assignment statement. Depending on the truth value of *cond*, the execution of the program will display or assign object R.

The next example shows the use of more alternatives and blocks with more statements:

```
DEF R: { A: integer, B: string }
DEF S: { C: integer }
IF MEMBER(3 ; S) THEN
[
    DISPLAY [] R
    R <- TU [ A := A + 1 ] R
]
ELSIF EX [ B = "alpha" ] R THEN
[
    DISPLAY[] S
    S <- PJ [ A ] R
    R <- SL [ A > 0 ] R
]
OTHERWISE
    R <- SL [ A > 0 ] R
```

In the next example the use of local objects and the specification of a conditional statement without the *otherwise* branch are shown:

```
DEF R: { A: integer }
DEF S: { B: integer }
IF EQUAL(R;{}) THEN
[
    DEF R: { A: integer }
    READ [ R ]
    DISPLAY[] JN [ A = B ] R S
]
```

Note that the defined local object R covers the visibility of the external homonymous object. Thus the internal join operation is performed between S and the local object R. Clearly, the existence of local objects is limited by the block where the object is defined.

This rule can be extended also to external static objects if we see the ALGRES program as embodied into a virtual external block (actually persistent objects can be explicitly removed before the end of the program but this case does not damage the generality of this assertion).

Since a conditional statement can be part of a sequence of block statements, it is possible to nest conditional statements and, consequently, blocks of statements as in the following example:

```
DEF R: { A: integer }
DEF S: { B: integer }
IF EQUAL(R;{}) THEN
[
    DEF R: { A: integer }
    READ [ R ]
    IF EX [ A > 0 ] R THEN
    [
        DEF S: [ C: string, D: integer ]
        READ [ S ]
        DISPLAY[] SL [ D > 0 ] S
    ]
    OTHERWISE
        DISPLAY [] S
]
```

Note that in the *IF* branch operand *S* corresponds to the local defined object of type multiset, while in the *OTHERWISE* branch *S* corresponds to the global object of type set.

6.1.2 Loop Statement

The loop statement semantics is similar to that of *while* statements of traditional structured programming languages: a condition is iteratively evaluated and, in case of true value, the block of statements is executed. Loop statement can constitute an alternative to the closure operator in the implementation of recursive algorithms. The following example shows a simple use of this statement:

```
DEF R: { A: integer }
READ [ R ]
WHILE EX [ A > 0 ] R DO
[
    DISPLAY[] R
    R <- TU [ A := A-1 ] R
]
```

You can see that the loop condition determines the existence of an element of R greater than zero. In that case R is displayed and then attribute A decremented. We expect that this loop will not be infinite since, after a certain number of iterations, all tuples of R will have a value less or equal to zero. The responsibility of the termination of the loop is left to the programmer (as it happens for the fixpoint operation).

In the next example we complicate the body of the loop by nesting another loop and introducing a local object:

```
DEF R: { A: integer }
READ [ R ]
WHILE EX [ A > 0 ] R DO
[
    DISPLAY[] R
    R <- TU [ A := A-1 ] R
    DEF S: [ B: integer ]
    S <- R
    WHILE AF [+/B] S > 10 DO
    [
        DISPLAY[] SL [ B > 1 ] S
        S <- TU [ B := B-2 ] S
    ]
]
```

In that case before of terminating the body of the outer *WHILE* statement a new local object S is defined and instanciated with the current value of the global object R. Then a new loop statement controlled by an aggregate operation is performed. The body of the nested loop is represented by the display of a selection of S followed by a tuple update operation which decrements the value of attribute B. Hence it is clear that also in this case the outer loop terminates since the nested loop behaves in a monotonic way at each iteration so that the condition expressed by the aggregate function will become true.

In the following example we show how to implement a typical recursive algorithm by means of a loop statement. The same algorithm is first implemented by means of the fixpoint operator:

```
DEF  persons:  {  person: string,  children:  {  child: string } }
DEF input: { person: string }
descendants <- FP [ temp := AF [ UNION/children  ]  NJ input persons;
                    CONTEQ(temp;input);
                    UN temp input
                  ]
               input
```

Note that the fixpoint operation generates the set of descendants of the input persons using iteratively the natural join and aggregate union operations. The same result can be obtained by the following loop statement:

```
DEF  persons:  {  person: string,  children:  {  child: string } }
DEF input: { person: string }
DEF result: { descendant: string }
WHILE NOT EQUAL(input;{}) DO
[
    input <- AF [ UNION/children] NJ input person
    result <- UN result input
]
```

It is noted that this implementation is not exactly the representation of the fixpoint algorithm in terms of loop statement since the result is "disconnected" from the input. This is possible because the parent-children relationship is represented by a tree and not by a generic graph. In this case, after an opportune number of steps, the input set will be empty, that is when the previous value of input is represented by the leaves of the tree. This entails that the repeated joins of the fixpoint implementation for the same persons in the combiner are avoided, thus providing a more efficient implementation of the algorithm.

6.2 Conditional Expression

The conditional expression generalizes the conditional paradigm introduced in selective tuple functions. A conditional expression generates a result depending on the truth value of a predicate. For each predicate a candidate expression is defined. Unlike selective tuple functions, the *otherwise* branch is mandatory, as shown by the syntax:

$$ConditionalExpr \rightarrow IfBranch\ (ElsifBranch)^{*}\ OtherwisBranch$$

Conditional expressions can be used whenever a (complex) value is expected, as for dynamic input. They are syntactically expressed like the *IfAssignment* of selective tuple functions but are not part of a specialized operator and, thus, can be used in different contexts. The resulting value corresponds to the first candidate expression which is related to a true condition. In the sequel we will write various examples corresponding to different evaluation contexts.

The first example is concerned with the use of a conditional expression to assign a value to a static object depending on given conditions:

```
DEF R: { A: integer, B: string }
DEF S: VS[R]
S <- IF EX [ A > 0 ] R THEN
```

```
    SL [ B = "alpha" ] R
  ELSIF EX [ A = 0 ] THEN
    SL [ B = "beta" ] R
  OTHERWISE
    R
```

Note that the value of S after the assignment depends on the truth value of conditions in the conditional expression. Each condition corresponds in our example respectively to the existence of a particular value of attribute A of R. Note that these conditions are correctly expressed since they respects the safety rule of predicates which prevents to have ambiguities in the evaluation of expressions. Note also that, as for selective tuple functions, each candidate expression shares a compatible schema (actually the same schema of R in the example).

The next example applies the conditional expression to the display operator. The reader can generalize it by replacing *DISPLAY* with other algebraic operators:

```
DEF R: { A: integer, B: string }
DEF S: { C: integer, D: string }
DEF cond: bool
DISPLAY [] IF cond THEN
             SL [ B = "alpha" ] R
           OTHERWISE
             SL [ D = "beta" ] S
```

In the example we used a simple boolean object to test the condition. This clarifies that the condition part is in general unrelated to the candidate expressions. Moreover, we used two different unrelated candidate expressions, one over R and one over S, to underline as well the orthogonality of candidate expressions.

In the next example, we show how conditional expressions can be nested. Note that the fact that the otherwise branch is mandatory entails avoids ambiguities in the syntax:

```
DEF R: { A: integer, B: string }
DEF cond: bool
DISPLAY [] IF cond THEN
             IF AL [ A > 0 ] R THEN
               SL [ B = "alpha" ] R
             OTHERWISE
               SL [ B = "beta" ] R
           OTHERWISE
             SL [ D = "omega" ] S
```

As a last example, let us consider the implementation of a selective tuple update in terms of a tuple update coupled with a conditional expression in the right part of the assignment:

```
DEF R: { A: integer, B: { C: string } }
R <- TU [ B := IF A > 0 THEN
                 SL [ C = "alpha" ] B
               ELSIF A = 0 THEN
                 SL [ C = "beta" ] B
               OTHERWISE
                 B
        ] R
```

In the above example we are forced to specify the default value in the *OTHERWISE* branch since we choose a tuple update *TU* and not a selective update *SU* which has a special default semantics. Selective tuple functions are to some extent redundant since they can easily be expressed in terms of normal tuple functions and conditional expressions. This is actually true, but for historical reasons conditional expressions were introduced in a successive extension of the ALGRES language. Specifically, the use of conditional expressions is fundamental in recursive subfunctions where the end of the recursion has to be expressed by a condition which is part of the subfunction body.

6.3 Subprograms

ALGRES code can be encapsulated into modules called subprograms which can be either subfunctions or procedures. Actually subfunctions were already part of the ALGRES language, but they have been extended to cope with stronger type-checking and recursion. Both subfunctions and procedures are defined in terms of a signature and a body. The signature specifies the kind of parameters of the subprogram, while the body represents the piece of algebraic code to be executed at the subprogram call.

This is conceptually similar to the usual subprograms in traditional general purpose programming languages. Thus, in the specification of the signature of the subprogram, parameters are called *formal*, as they are instanciated in one ore more calls with actual parameters, that is the actual instances associated to the subprograms. Note that, unlike general purpose languages, parameters of ALGRES subprograms are in general complex objects. This difference is automatically applied to bodies which include complex algebraic expressions.

The main difference between a subfunction and a procedure is that the former encapsulates an expression computing a (complex) value, while the latter encapsulates a list of statements to be sequencially executed. This distinction implies that, unlike procedures, subfunctions can be used as operands in ALGRES expressions, for example as operand of a *DISPLAY* operator.

Another important difference is concerned with the kind of parameters. In subfunctions parameters are passed only by value and cannot be affected by the execution of the body. Conversely, procedure's parameters can be passed either by value or by reference. Moreover, both by value and by reference parameters can be affected in the body of the procedure, albeit the changes are visible outside the procedure (after the procedure call) only for parameters passed by reference.

The possibility of altering the instance of parameters in procedures stems from the fact that the body of a procedure consists of a sequence of statements among which assignments of parameters with new values might hold. Note that this is exactly what happens in traditional programming languages like Pascal.

Another consequence is that bodies of procedures may include the definition of local objects whose life is embraced by the block where they are defined. This naturally follows from the observation that the body of a procedure is actually a block similar to those which can be defined in control statements.

A common feature of subfunctions and procedures is the lack of external visibility. This means that formal parameters and, in case of procedures, possibly the defined local objects are the only visible objects within the body. This fact is motivated by the choice of having non-contextual subprograms. In operative terms this can be rephrased as follows: the definition of the body of a subprogram is independent from the context where it is specified. On the other hand it is posible to define the schema of parameters by means of a *VS* vide-supra reference to external objects.

A peculiar characteristic of both subfunctions and procedures is recursion. Recursion is a valid alternative to closure operator in the definition of a certain intrinsically recursive algorithms. A recursive call of a subprogram holds whenever the same subprogram is called in the execution of the corresponding body.

In that case the ALGRES interpreter follows the usual stack policy of record activation: at the nested (recursive) call, a copy of the by value and local parameters is accomplished. Before of using the name of a subprogram, it must be defined the definition of the corresponding signature. This could be a problem when there is a mutual recursion between two subprograms. In that case, it is possible in a first step to declare the signature of the subprogram (without the body) and, in a forthcoming step, to declare signature (actually in this case the signature is redeclared) and body, that is the complete definition of the subprogram.

6.3.1 Subfunctions

The subfunction declaration consists of two parts, a specification and a body. The translator checks that the body uses parameters in a way compatible with the pecification and that the subfunction name does not already exist. The definition part specifies the structure of the formal parameters, which are the only visible operands within the body:

$subfunction \rightarrow SUBFUNC\ SubfuncDecl\ SubfuncBody$
$SubfuncDecl \rightarrow identifier\ FParameterList : ResultType$
$FParameterList \rightarrow (DefList \mid \epsilon)$
$ResultType \rightarrow TypeExtension$

$SubfuncBody \rightarrow [function]$

The result of a subfunction has no name; in order to use it in a program it is necessary to call the subfunction explicitly or to assign the subfunction result to another object. The correspondence between formal and actual parameters is given by means of their position. Each actual parameter must be of compatible type of the corresponding formal parameter.

In the following example we define a subfunction called *WithName*, computing the number of persons having a given name:

```
SUBFUNC  WithName(People:{ name: string, age: integer}, Name: string):
{name: string, age: integer}
[
    CARD ( SL [ name = Name ] People )
]

DEF  Persons: { N: string, A: integer }
Jhon <- WithName(Persons, "Jhon")
Jhon-or-Mary <- WithName(Persons, "Jhon") + WithName(Persons, "Mary")
```

It is possible to nest some function calls because actual parameters can be represented by the object resulting from any operation.

The next example rephrases the descendant problem, previously expressed by means of the loop statement, in terms of a recursive subfunction:

```
DEF  persons:  {  person: string,  children:  {  child: string } }
READ [ persons ]

SUBFUNC descendants(input: { person: string }, persons: VS[]):
{ person: string }
[
    IF EQUAL(input; {}) THEN {}
    OTHERWISE
     UN input descendants(AF [ UNION/children ] NJ input persons, persons)
]

DISPLAY[] descendants({("Paul")}, persons)
```

The signature of the subfunction consists of parameters *input* and *persons* plus the output result which is the set of descendant persons. The body is represented by a conditional expression: if the input is empty, then so it is the result, otherwise the result is composed of the union of the input and the recursive call of descendants on the children of input.

This new value is computed by the aggregate function *AF* which makes the union of all the children of input persons.

Note that after the subfunction specification we wrote a display statement whose operand is the subfunction call applied to the input singleton set represented by the person *Paul*, and the set of persons.

We might call the subfunction and use the result as operand of an algebraic operator as in the following case:

```
DEF name: string
READ [ name ]
exists-name <- EX [ person = name ] descendants({("Paul")}, persons)
```

In that case *exists-name* will contain after the call of descendants a boolean value representing the fact that the given name is included or not in the set of the descendant persons of *Paul*.

The reader can see that the use of subfunctions is similar to that of algebraic operators and, actually, subfunctions can be seen as non-polymorphic user-defined n-ary operators. Non-polimorphism derives from the fact that, unlike algebraic operators where the schema of operands can be of several (actually infinite) types, parameters of subfunctions are statically defined in terms of their schema.

6.3.2 Procedures

Procedures are a means of encapsulating a piece of ALGRES code as a sequence of statements. As subfunctions, they are defined by a *signature* and a *body*. The former specifies the schema of the formal parameters, while the latter represents the actual code to be executed when the procedure is called.

Unlike subfunctions, procedures allows to declare parameters either *by value* or *by reference*. Both kinds of parameters can be assigned in the body of the procedure with new values. In case of parameter by reference, this possible new instantiation will cause a side effect on the corresponding actual parameter. Moreover, a procedure call cannot be used as a value in algebraic expressions, as it corresponds to a new *CALL* statement.

Like subfunctions, procedures can be recursively defined. The defintion of the signature of the procedure must always precede its call. In case of mutual recursion between procedures *P1* and *P2*, it is possible to solve the dilemma of how properly define the procedure by first declaring the signature of *P1* without specifying the body. Then it is possible to define the signature and the body of *P2*. Finally, a complete definition of *P1* (involving both signature and body) can be performed. This implies that in general it is possible to re-declare the signature of the procedures (and subfunctions) more than once. Obviously these multiple declarations must be compatible each others.

The syntax of procedure declaration and call is as follows:

$procedure \rightarrow PROCEDURE\ ProcedureDecl(\epsilon | ProcedureBody)$
$ProcedureDecl \rightarrow ProcedureName\ ProcParameterList$
$ProcParameterList \rightarrow ProcedureName(ProcDefList \mid \epsilon)$
$ProcedureName \rightarrow identifier$
$ProcDefList \rightarrow procDefDescr(,\ procDefDescr)^*$
$procDefDescr \rightarrow identifier\ (ByValue \mid ByReference)\ TypeExtension$
$ByValue \rightarrow\ :$
$ByReference \rightarrow -\!>$
$ProcedureBody \rightarrow [BlockStatement]$
$ProcedureCall \rightarrow CALL\ ProcedureName(AParameterList)$

The only visible objects inside the body of a procedure are the formal parameters and the local objects possibly defined in the block corresponding to the body.

The qualification of the kind of parameter (by value or by reference) is accomplished by means of the special character following the name of the parameter. A normal definition, that is a definition of the type *parameter: schema* represents a by value parameter. In case of by reference parameters, the colon character is replaced by the right-arrow. Thus the general form of a parameter declared as passed by reference is *parameter→schema*.

This lexical subtle distinction entails different mechanisms. First, by reference actual parameters mut always be object names, while by value actual parameters can be general expressions. Second, albeit both by value and by reference parameters can be modified by the execution of the body, only by reference parameters maintain the new instance outside the scope of the procedure. In other terms, side effects are only possible to parameters passed by reference.

Note that not all kinds of statement can be written in the body of procedures. Specifically, statements concerning persistency management, such as creation or removal of persistent objects or groups, are disallowed, as well as to open or close transactions.

Apart form this class of statements, statements in the body of procedures can be freely used. Actually it is possible to use all the statements which can appear in a block. In particular it is possible to specify control statements and procedure calls. This last opportunity opens the possibility to define recursive procedures.

In the following example the problem of determining the free rooms of a school in a given day and hour is solved by calling a declared procedure:

```
DEF courses:
{
    course-name: string,
    sections:
    {
```

```
        section: char,
        teacher: string,
        time-table:
        {
            day: string,
            hour: integer,
            classroom: string
        }
    }
}

PROCEDURE DisplayFreeRooms
(courses: VS[], free-day: string, free-hour: integer)
[
   rooms-time <-
       CS [ room-time: { course-name, section, teacher, day, hour } ]
       DS [ sections ]
       DS [ time-table ] courses

   DEF free-rooms: { free-room: string }

   free-rooms <- PJ [ classroom ]
                 SL [ NOT EX [ (day = free-day) AND
                               (hour = free-hour) ] room-time
                    ] rooms-time

   DISPLAY [] free-rooms
]

CALL DisplayFreeRooms(courses, "Thursday", 9)
```

In the above example the *DisplayFreeRooms* procedure is declared with three parameters passed by value. The first parameter *courses* corresponds to the object on which is performed the computation. This must be declared since no other external objects are visible from the body of a procedure. The other two parameters represent respectively the day and the hour of the timetable for which the free room is searched. These formal parameters are instanciated at the procedure call by *Thursday, 9*.

Note that the body of the procedure is composed of four statements. The first statement creates by instanciation a new local object *rooms-time* corresponding to the timetable of each classroom. The creation of this object is not preceded by a corresponding schema definition. The second local object *free-rooms* is explicitly created by a *DEF* statement and is instanciated by the third statement which implements the central part of the procedure, that is the computation of the free rooms. The last statement displays the found free rooms. The execution of the statement after the procedure declaration correponds to the procedure call applied to the actual parameters. This will cause the visualisation of the free rooms at the given time.

The next example shows the use of a procedure with parameter passed by reference. This is useful whenever we want to make an update on a given object by means of a subprogram. The example refers to object *courses* defined above. The update changes the classrooms of lessons of a given teacher into a new given room:

```
PROCEDURE ChangeClassRoom(courses -> VS[])
[
    DEF teach: string
    DEF newroom: string
    READ [ teach ]
    READ [ newroom ]
    courses <- SU [ classroom := IF teacher = teach THEN newroom ]
                courses
]
```

In this example only one parameter is passed (by reference) to the procedure. The names of the teacher and the new classroom are dynamically determined by the execution of the procedure. They correspond to local objects. The last statement of the body updates the object *courses* by assigning to the given teacher the chosen new room.

Note that in this implementation there is no check on the referential integrity of the input values. In other terms, the procedure does not test the membership of the given teacher and classroom in the database. Thus, we modify the above procedure in order to guarantee a correct update. For this reason we suppose to have two more objects *teachers* and *classrooms*:

```
DEF teachers: { name: string, phone: integer }
DEF classrooms: { room: string, dimension: integer }

PROCEDURE ChangeClassRoomRevised
(courses -> VS[], teachers: VS[], classrooms: VS[])
[
    DEF teach: string
    READ [ teach ]
    WHILE NOT EX [ name = teach ] teachers DO
    [
        DISPLAY [ "wrong teacher" ] teach
        DISPLAY  ["correct  teachers"] PJ [ name ] teachers
        READ [ teach ]
    ]

    DEF newroom: string
    READ [ newroom ]
```

```
    WHILE NOT EX [ room = newroom ] classrooms DO
    [
        DISPLAY [ "wrong classroom" ] newroom
        DISPLAY  ["correct  rooms"] PJ [ room ] classrooms
        READ [ newroom ]
    ]

    courses <- SU [ classroom := IF teacher = teach THEN newroom ]
               courses
]
```

The new procedure is a little more complicated as it loops for the input instances until the given value is correct. This condition is implemented by the existence operator applied respectively to *teachers* and *classrooms*. In case of wrong input, an error message is displayed with the visualization of the wrong value. Furthermore, the list of the correct values is given to the user by means of another display.

The next example represent yet another implementation of the descendants problem previously solved using two different styles, the loop statement and the recursive subfunction. In this example we rephrase the latter implementation in terms of the procedural paradigm:

```
DEF  persons:  {  person: string,  children:  {  child: string } }
READ [ persons ]
DEF result: { person: string }
result <- {}

PROCEDURE descendants(input: VS[result], persons: VS[], result -> VS[])
[
    IF NOT EQUAL(input; {}) THEN
    [
        CALL descendants(AF [ UNION/children ]
                         NJ input persons, persons, result)
        result <- UN result input
    ]
]

CALL descendants({("paul")}, persons, result)

DISPLAY[] result
```

Note that the procedural implementation is less concise than the subfunctional one. Nethertheless this procedure is to some extent isomorphic to the corresponding subfunction implementation. The difference lies in the fact that the name of the procedure cannot be directly used as a value in the algebraic expressions. This is why we inserted a new object result among the parameters of the procedure. This parameter is passed by reference and represents the partial result of the computation. The recursive call of descendants is explicitly performed by means of the *CALL* statement.

6.4 Links

The ALGRES language allows to define a special kind of relationship between attributes belonging to different objects called link relationship, or more simply link. Links can be specified on a schema of defined static (possibly persistent or local) objects.

Loosely speaking, a link between an attribute A of object R to an attribute B of object S specifies a path which allows the ALGRES programmer to pass from the current instance of attribute A to the instances of attributes at the same quantification level of B in object S. In the context of a link between R and S, the former (on the left) is called *base collection* while the latter (on the right) is called *target collection*. Analogously, attribute A of R is called the *base attribute*, while attribute B of S is called *target attribute*.

$LinkStatement \rightarrow LINK\ LinkName\ [PathIdent -> \ PathIdent]$
$LinkName \rightarrow identifier$
$AttributeName \rightarrow identifier$
$LinkPath \rightarrow LinkName-> (LinkName->)^{*}\ AttributeName$

Links can be defined only between simple attributes. Normally links are defined between attributes which share the same semantics, i.e. which correspond to the same concept in the real world modelled by the database schema. Another interpretation of the link concept is the virtualization of a given schema by means of new (virtual) attributes corresponding to the linked attributes in the other objects.

Before of using a link, it must be defined. Once defined, a link allows to write more concise algebraic code and consequently to speed the interpretation. The link concept, however, as it happens for control statements, does not enrich the computational completeness of the language. Nethertheless it might represent a good solution for applications involving an intensive attribute-relationship semantics.

Hereafter we will give first a list of abstract examples aiming at showing the main capabilities of the link concept. At the end of the section we will be concerned with more realistic applications. The first example represents the definition of a link between two flat collections:

```
DEF R: { A: char, B: integer }
DEF S: { C: integer, D: string }
LINK L1 [ R.B->S.C ]
DISPLAY ["result"] PJ [ A ] SL [ L1->D = "anne" ] R
```

The above ALGRES code defines sets R and S, and links attribute B of R to attribute C of S. Note that B and C share the same simple type integer.

Links can be defined also for the other simple types (char, string, real, and boolean). In any case, the two involved attributes must be simple and be defined on the same type.

Moreover, the attribute on the right (C in our example) must always be placed at the outer quantification level, i.e. it must belong to the tuple of the collection. On the other hand, the attribute on the left (B in our example) can be at any level. The rationale of this rule stems from the fact that we want links to provide a mechanism for associating to an attribute a tuple of another object.

We are interested in fetching attributes of tuple of S without making joins between R and S. Once defined link *L1*, it is possible to select tuples of R by evaluating expressions involving attributes of S without joins. In our example the selection predicate is composed of an equality between a link-path ($L1 \rightarrow D$) and a string constant (*anne*). This comparison is allowed since link *L1* provides a means of evaluating attribute D of S starting from the current value of attribute B of R. In practice, for each tuple of R (and consequently for each value of attribute B) the interpreter will search a tuple of S such that $R.B = S.C$. Then, the correponding attribute D of S is returned as result of the link-path operator. Note that in general more than one tuple of S might correspond to a single value of B in R. In that multi-values case, the interpreter will return indeterministically one value of C among all possible values. We are now ready to give a list of rules representing a synthesis of our discussion on the correct use of links:

Remark: Given a link $A1 \rightarrow A2$ between attributes respectively belonging to object R and S, the following conditions must hold: i) R and S are collections; ii) *A1* and *A2* are simple attribute having the same type; iii) the quantification level of *A2* (the target attribute) is that of tuples of S (the target collection).

Note that we might obtain the same result by rewriting the above code using the traditional join operator as follows:

```
DISPLAY  ["result"] PJ [ A ] JN [ (B = C) AND (D = "anne") ] R S
```

In the next example we define a link between more complex objects and we will use the link-path notation to fetch a complex attribute of the target collection:

```
DEF R: { A: char, B: { C: integer } }
DEF S: { D: integer, E: { F: string } }
LINK L1 [ R.B.C->S.D ]
DISPLAY[] SL [ EQUAL(L1->E;{}) ] R
```

In that case, the base attribute C is nested in the base relation R, while the target attribute D is at the qunatification level of tuple of S (as required). Moreover, the selection predicate involves a link-path expression fetching the complex atttribute *E* of the target collection *S*. The predicate is true if the fetched instance of *E* is an empty set.

Note that in this case the quantification level of the selection operator corresponds to *C* (nested predicate). This means that the selection will operate on the nested collections *B* of *R*.

The following example shows the use of multiple links in link-path expressions. This capability represents a natural extension of pathnames, with the difference that link-paths span over different objects:

```
DEF R: { A: char, B: integer }
DEF S: { C: integer, D: string }
DEF T: { E: string, F: integer }
LINK L1 [ R.B->S.C ]
LINK L2 [ S.D->T.E ]
DISPLAY[] SL [ L1->L2->F = 20 ] R
```

In the above ALGRES code three objects and two links are defined: *L1* links integer attributes *B* and *C*, while *L2* links string attributes *D* and *E*. The selection predicate is represented by a path-link involving both *L1* and *L2*. This means that for each tuple of *R*, the link-path generates a value generated following the chain indicated by links *L1* and *L2*.

A more complicated scenario can be obtained by defined links with cycles. Indeed unlike pathnames, the definition of multile links over a set of objects might result in a cyclic graph. This situation arises for example when a link *L1* is defined from object *R* to object *S*, and a link *L2* is defined from *S* to *R*. In such cases, the involved link-path expressions may assume an indefinite length, as showed in the next example:

```
DEF R: { A: char, B: integer }
DEF S: { C: integer, D: string }
DEF T: { E: string, F: integer }
LINK L1 [ R.B->S.C ]
LINK L2 [ S.D->T.E ]
LINK L3 [ T.F->R.B ]
DISPLAY[] SL [L1->L2->L3->L1->L2->F = 4] R
```

The above example represents a modification of the previous one where a new link *L3* is defined between *T* and *R*. Note that *L3* entails a cycle in the link-path schema involving *R*, *S* and *T*. This situation enables the possibility of writing more complex link-path expressions in the selection predicate.

The next example is concerned with a concrete situation where *persons*, *students* and *courses* are involved:

```
DEF Persons: { name: string, city: string, phone: integer }
DEF Students: { name: string, school: string ,
                courses: { course: string } }
DEF Courses: { name: string, teacher: string }
```

In the picture we linked the name of a student with the name of a person, meaning that a student is a person (*ISA* relationship). Similarly we linked attribute course of *Students* to attribute name of object *Courses*, meaning that the string represented by the base attribute corresponds to a course name in the target attribute. Finally, we linked attribute teacher of *Courses* to attribute name of *Persons*. This establishes that teacher represents the name of a person. These relationships can be formilized in the ALGRES schema in terms of the folowing code:

```
LINK StudPers [ Students.name->Persons.name ]
LINK StudCour [ Students.Courses.course->Courses.name ]
LINK CourPers [ Courses.teacher->Persons.name ]
```

We are now ready to make queries on the enriched schema using link-path expressions. For example we might be interested in finding the students attending at least one course teached by a professor living in the same city of the student:

```
DISPLAY []
SL [ EX [StudPers->CourPers->city = StudPers->city ] courses ]
Students
```

In the above expression the selection predicate is represented by the comparison of two string value returned by correponding link-paths. Th first link-path expression ($StudPers \rightarrow CourPers \rightarrow city$) fetches the city of the professor teaching the current course of the current student (note that for each student there are several courses).

The second link-path expression ($StudPers \rightarrow city$) simply returns the city of the current students. Note that both link-paths find the city in the same target object *Students*, but this does not entail that the two cities are the same since paths are different. This is clear if we think that the first link involves the name of a student, while the second one involves the name of a teacher, althought both names are targetted on names of persons.

6.4.1 Dealing with unmatching links

The *MATCH* built-in function, is introduced to cope with undefined values returned by link-path expressions. *MATCH* has a chain of link names as argument and returns a boolean value which is true if there exits a value of the target attribute matching the cuurent value of the base attribute, otherwise it returns false.

MATCH function is useful whenever using a link-path expression we are not sure about the actual existence of a target attribute matching the current value of the base attribute. In such cases the ALGRES interpreter will return an indefinite value. For example, given the following example:

```
DEF R: { A: char, B: integer }
R <- {('a', 3)('b', 2)('c', 4)('d', 1)}
DEF S: { C: integer, D: string }
S <- {(1, "anne")(2, "mary")(3, "john")}
LINK L [ R.B->S.C ]
DISPLAY[] SL [ L->D = "mary" ] R
```

In that case the interpreter will return for the fourth tuple and indefinite value. This situation should be avoided in order to be sure about the correct interpretation of the selection. For case like the above one, in which the application does not guarantee the correct matching of link-paths for all the involved tuples, it is recomanded to use *MATCH* function. So the above code can be conveniently transformed into the following one:

```
DISPLAY [] SL [ IF MATCH(L) THEN L->D = "mary" OTHERWISE false ] R
```

The rationale of the above statement can be described as follows: since, due to a possible unmatching between base attribute B of object R and the target attribute C of object S, we are in general not sure about the actual value returnded by the link-path expression $L \rightarrow D$, we define the selection predicate as a conditional expression depending on the fact that the link-path is defined (there exists matching between attributes) or not. If it is defined, the predicate expression coincides with the natural predicate, otherwise a false boolean value is forced.

Note that the need to correctly manage an unmatching link-path is not only concerned with selection predicates but regards every algebraic expression where a link-path is involved. Thus, this problem might be solved for the following statement as well (we suppose it is applied to the above schema and instances):

```
DISPLAY[] TE [ N := L->D ] R
```

This tuple extension builds a new attribute N whose value is represented by a path expression. As for the above selection case, the programmer should be aware about the indefinite value for some tuples (the fourth unmatching tuple in the example). In order to handle this exception he or she might assign a precise value for these tuples as follows:

```
DISPLAY [] TE [ N := IF MATCH(L) THEN
                       L->D
                     OTHERWISE
                       "unmatching name"
                ] R
```

It is possible to use an arbitrary multiple link as argument of the *MATCH* function, as in the following example:

```
DEF R: { A: char, B: integer }
R <- {('a', 3)('b', 2)('c', 1)}
DEF S: { C: integer, D: string }
S <- {(1, "anne")(2, "mary")(3, "john")}
DEF T: { E: string, F: integer }
T <- {("mary", 10)("anne", 20)("david", 30)}
LINK L1 [ R.B->S.C ]
LINK L2 [ S.D->T.E ]
DISPLAY[] TE [ N := L1->L2->F ] R
```

This case rises an exception for the first tuple of *R* since link *L2* unmatches the value john of base attribute *D* with a corresponding value of target attribute *E*. We could rewrite the tuple extension expression by including as in the previous example a conditional expression governed by the *MATCH* function:

```
DISPLAY [] TE [ N := IF MATCH(L1->L2) THEN
                        L1->L2->F
                     OTHERWISE
                        -1
                ] R
```

7 CONCLUSIONS

Contributed by: S. Ceri

The design and implementation of the Year-2 ALGRES testbed has been a significant effort so far, due to the intrinsic complexity of the problem, the small amount of time available, and the the size and variety of the POLI team of designers and implementors (including an overall number of 5 faculty members, 3 PhD students, 2 senior researchers, and 19 master-level students, though at various degree of involvment).

Coordination has been ensured by establishing a strong project discipline: each group has been encouraged to spend efforts in the specification and design, with particular care placed on interface documents delimiting the activities of each group. Our implementation exhibits a classical "chain" organization of software modules, where each software module produces as output the input to the next level in the chain; we have overcome the difficulties due to parallelism by an early development of case studies which covered the entire testbed, so that each group had available, rather early, "manually produced" outputs of the previous group.

The project has already indicated several results that will be quite useful for replicating an implementation of triggers in an industriel setting:

1. The set-oriented nature of triggers in Chimera has proved to be useful for enabling the compilation (and optimization) of triggers, which give rises to code for triggers that runs separately from the code of transactions. In particular, optimization techniques for the condition part of triggers are the same used for declarative expressions of the *select* query primitive.

2. The early evaluation of event predicates has proved useful for immediately suspending the evaluation of a rule condition, if they are empty. This built-in optimization reinforces the design criterion that triggers should be as much as possible *incremental*, i.e. they should use state changes (deltas) in their condition whenever possible [10,11].

3. In deciding how to compute the event instances, we have been faced by a rather difficult trade-off analysis between doing most work at compile time or doing some of it at run-time. As result of this analysis, most of the work (including the computation of events propagation due to generalization hierarchies) is done at compile time; however, the computation of event predicates from event instances was left to the run-time component. This analysis was influenced by the fact that at compile time we master (general-purpose, sub-optimal) ALGRES structures, while the run-time system can make use of special-purpose optimized data structures (the *event tree*). More in general, we observe that even more substantial modifications of a general-purpose algebraic machines might be essential in order to improve performances; such modification should automatically and transparently propagate event instances and store them into the appropriate collections. Something similar is performed by *attachments* (tuple-level procedures performing event instance management at each SQL operation on a tuple) in the implementation of Starburst [13].

4. We have designed a preliminary version of a user-friendly interface that supports already a rather sophisticated user interaction. When the debugging mode is selected, the user may choose to perform step-by-step rule execution and inpect the content of the most significant run-time features (such as the conflict set at each rule consideration). We also enable the user to choose one of the triggers at the same level of priority and impose it for execution before the other ones. In this way, the user may perform simple analysis of how the rules behavior is modified as effect of a change of priorities. We believe that after experimentation with the user interface it will become possible to design additional rule manipolation commands that may dynamically alter their properties in the context of a design session.

5. Preliminary measures of performance of triggers wrt transaction lines are very encouraging: trigger selection is very fast and thus the overall execution times are dominated by execution time of ALGRES code respectively produced for transactions and for triggers. These two execution times, in turn, depend very much on the application (in particular, of the degree of processing which is assigned to triggers as opposed to that of transactions); we observe, for instance, that in the EMS application trigger execution dominates over transaction execution, but this is quite expected by considering that the transaction performs a single change on the power of a node (accessed with a selective condition), and next trigger propa-

Productions	Attributes	Semantic RUles	Semantic Functions
90	22	206	44

Figure 19: DD-Module : Statistics

gate that change along the network on an arbitrary number of predecessors of that node.

At this stage (may 20) the project is far from being completed, and although promising intermediate results have been achieved, we are still testing the integration of the various software modules which cooperate to the testbed architecture. We expect that most of the relevant features of the testbed will be demonstrated in the project review (June 16-17), and that a consolidated version of the testbed will be available for export to other Consortium partners (SEMA, ICRF) in autumn; this will be also the time when most of the students participating to the project will be graduated.

A1 GRAMMAR OF THE YEAR-2 ALGRES TESTBED

Contributed by: D. Berveglieri, M. Campriani, C. Cappellini, C. Draxler, R. Penzo, G. Psaila

A1.1 DD Module

The DD module is rather large, but was not split into smaller s-modules because during year-1 the federal grammar approach was still under development; moreover its dimension is not so large and the compiler generation process for this module does not require so much time.

The semantics associated to this module uses the Vichi-1 library to populate the *Data Dictionary* with new class schemas provided by the user in Chimera language. Several properties of a new schema are checked against the existing schema or for internal coherence; the compilation fails if at least one of the checked properties does not hold. When the compilation fails, the transaction is not aborted, but the class which caused the failure is not saved in the Data Dictionary after a *commit* statement execution. In such case, no target code is generated.

The semantic evaluator visits the syntax tree, generated by the parser, following the so called *one-sweep left-to-right strategy*, that is it proceeds from left-to-right and top-down; this visit strategy guarantees a high efficiency of the evaluator, and, as consequence, of the whole translator.

Table 19 briefly resumes the four dimensions characterizing this grammar module.

A1.1.1 Syntax Productions

```
schema ::= "begin" "transaction" ";" transaction_body end_transaction ";"

end_transaction ::= "commit" | "rollback"

transaction_body ::= ddl_transaction

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

ddl ::= 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"
```

A1.2 TF Module

The TF module (Terms and Formulas) is rather large, but has not been split into smaller s-modules because the s-modules obtained by splitting it would be mutually recursive :

Productions	Attributes	Semantic RUles	Semantic Functions
147	52	785	110

Figure 20: TF-Module : Statistics

in Chimera a *term* can need a formula (for instance an aggregate operator and its *where* part) and we could split the TF-module into a Term module and a Formula module, where Term uses Formula; a formula is composed by means of terms, then the Formula module uses the Term module, that is they are mutually recursive. As consequence the TF-module is very large and the compiler generation process needs a lot of time and computing resources to generate the seemantic evaluator (at least in the actual version of tools which are part of our GRAMMATICA environment).

The semantics associated to this module uses the Vichi-1 and Vichi-2 libraries to get data from the *Data Dictionary* about previously defined classes (Vichi-1 services) and translate terms and formulas into the intermadiate language IGLOO (by means of Vichi-2 services). This intermadiate language allows us to decouple the analysis process of Chimera source code from the final translatioin into the Algres programming language.

Another important task of this module is type checking. In a Chimera formula variable definition can follow variable use, then the type checking process must be done only when every term in the formula has been analysed. In order to divide the semantic analysis in two phases, the semantic evaluator visits the syntax tree, generated by the parser, following the so called *multi-pass left-to-right strategy*, that is it visits the tree many times, proceeding from left-to-right and top-down at each pass; this visit strategy is not very efficient, but in our case only two passes are needed and the general efficiency of the evaluation is rather acceptable.

Table 20 briefly resumes the four dimensions characterizing this grammar module.

A1.2.1 Supported Productions for Terms

```
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

set ::= "{" [ 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 ] ")"
       | constant
       | start_with_variable
       | start_with_identif
       | aggregate_operator

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

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

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"

constant ::= "null"
/* the previous production is supported only for OIDs */
```

A1.2.2 Not Supported Productions for Terms

```
mult_op ::= "//"
```

```
set_or_list ::= list

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

simple_term ::= list_unary_operator

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

A1.2.3 Supported Productions for 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
                 | comparison_formula
                 | membership_formula
                 | type_formula

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

comparison_formula ::= expression comparison_op expression

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

membership_formula ::= expression "in" expression

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

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 ")"
```

Productions	Attributes	Semantic RUles	Semantic Functions
48	29	109	39

Figure 21: QUAC-Module : Statistics

```
        | "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_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 ::= [class_name "."] t_event { "," [class_name "."] t_event }

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

A1.2.4 Not Supported Productions for Formulas

```
atomic_formula ::= choose_formula

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

choosing_set ::= expression

cardinality ::= expression
```

A1.3 QUAC Module

The QUAC module is a small grammar module. It defines the structure of actions that constitute transaction lines or the action part of triggers. An action can be either a query or an update; in the latter case the QUAC module uses the UP module, otherwise in the former case queries are defined in the QUAC module itself.

This module uses the Vichi-2 library to translate Chimera code into an IGLOO representation; this intermadiate representation is translated by Vichi-2 into the target ALGRES programming language.

The semantic evaluator visits the syntax tree, generated by the parser, following the so called *one-sweep left-to-right strategy*, that is it proceeds from left-to-right and top-down; this visit strategy guarantees a high efficiency of the evaluator, and as consequence of the whole translator.

Table 21 briefly resumes the four dimensions characterizing this grammar module.

A1.3.1 Supported Productions

```
action ::= query_cmd
         | update
         | name_definition
```

A1.3.2 Not Supported Productions

```
action ::= operation_call
         | procedure_call

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"
```

A1.3.3 Supported Productions

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

```
display_cmd ::=
   "display" "(" start_with_var_or_id_list [where_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  "where" formula ")"

var_list ::= variable_name { ","  variable_name }
```

A1.3.4 Not Supported Productions

```
query_cmd ::= for_each_cmd

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

list_of_actions ::= action { connector action }

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

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

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

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"
```

Productions	Attributes	Semantic RUles	Semantic Functions
21	22	116	32

Figure 22: UP-Module : Statistics

A1.4 Update Primitives (UP Module)

The UP module is a small grammar module. It defines the structure of updates, a primitive occurring in transaction lines or in the action part of triggers.

This module uses the Vichi-2 library to translate Chimera code into an IGLOO representation; this intermadiate representation is translated by Vichi-2 into the target ALGRES programming language.

The semantic evaluator visits the syntax tree, generated by the parser, following the so called *one-sweep left-to-right strategy*, that is it proceeds from left-to-right and top-down; this visit strategy guarantees a high efficiency of the evaluator, and as consequence of the whole translator.

Table 22 briefly resumes the four dimensions characterizing this grammar module.

A1.4.1 Supported Productions

```
update ::= create_cmd
         | create_tmp_cmd
         | delete_cmd
         | change_persistency_cmd
         | generalize_cmd
         | specialize_cmd
         | modify_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 ")"
```

Productions	Attributes	Semantic RUles	Semantic Functions
41	28	98	39

Figure 23: TRIG-Module : Statistics

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

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

var_or_oid_name ::= oid_name | variable_name
```

A1.4.2 Not Supported Productions

```
update ::= population_cmd

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

A1.5 TRIG Module

The TRIG module is the grammar module which defines the structure of triggers; it is a top-module, then its axiom is the so-called entry point to the sublanguage of triggers (sublanguage recognized by the TRIG Compiler).

This module uses the Vichi-2 library to translate Chimera code (trigger structure aspects) into an IGLOO representation; this intermadiate representation is translated by Vichi-2 into the target ALGRES programming language.

The semantic evaluator visits the syntax tree, generated by the parser, following the so called *one-sweep left-to-right strategy*, that is it proceeds from left-to-right and top-down; this visit strategy guarantees a high efficiency of the evaluator, and as consequence of the whole translator.

Table 23 briefly resumes the four dimensions characterizing this grammar module.

```
tr_def ::= transaction {transaction}
```

```
transaction ::= "begin" "transaction" ";" transaction_body end_transaction ";"

end_transaction ::= "commit" | "rollback"

transaction_body ::= ddl_transaction

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

ddl ::= targeted | untargeted

targeted ::= implementation

implementation ::= targeted_trigger

targeted_trigger ::= targeted_trigger_rule

untargeted ::= untargeted_trigger

untargeted_trigger ::= untargeted_trigger_rule

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_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

reactions ::= reaction { connector reaction }

reaction ::= action
           | "savepoint"
           | "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
```

Productions	Attributes	Semantic RUles	Semantic Functions
27	7	18	11

Figure 24: DML-Module : Statistics

A1.6 DML Module

The DML module is the grammar module which defines the structure of transactions which manipulate data, the so-called Data Manipulation Language (DML); it is a top-module, then its axiom is the so-called entry point to the sublanguage of transactions (sublanguage recognized by the TRANS Compiler).

This module uses the Vichi-2 library to translate Chimera code (trigger structure aspects) into an IGLOO representation; this intermadiate representation is translated by Vichi-2 into the target ALGRES programming language.

The semantic evaluator visits the syntax tree, generated by the parser, following the so called *one-sweep left-to-right strategy*, that is it proceeds from left-to-right and top-down; this visit strategy guarantees a high efficiency of the evaluator, and as consequence of the whole translator.

Table 24 briefly resumes the four dimensions characterizing this grammar module.

```
trans ::= transaction {transaction}

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

end_transaction ::= "commit" | "rollback"

transaction_body ::= dml_transaction

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

dml ::= action {connector action} | "savepoint"

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

REFERENCES

[1] S. Ceri, R. Manthey: *Cosolidated Specification of Chimera (CM and CL)*, IDEA.DD.2P.006.01, November 1993.

[2] P. Dechamboux, P. Valduriez: *Architecture of the Peplom Compiler*, IDEA.DD.5B,

April 1993.

[3] A. Sikeler: *The Architecture of the ECLiPSe Testbed*, IDEA.WP.3E.010, February 1993.

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

[5] G. Lamperti: *Programming in ALGRES*, June 1992.

[6] F. Cacace, S. Ceri, L. Tanca: *Design and Implementation of Logres*, POLI Rep. 92.040, submitted for publication (June 1992).

[7] F. Cacace, *Integration of deductive and object oriented databases: theory and implementation* (PhD thesis, in Italian), POLI Rep. 92-039.

[8] E. Baralis, S. Ceri, J. Widom: *Better Termination Analysis for Active Databases*, Int. Workshop on Rules, August 1993.

[9] R. Manthey: *Some "heretical" ideas on designing DOOD systems*, oral presentation, minutes of the WP3 Meeting held in Manchester, IDEA.MA.3E.012, March 1993.

[10] S. Ceri, J. Widom: *Deriving Production Rules for Incremental Materialization of Stratified Datalog*, Rept. n. 92-081, Dipt. Elettronica e Informazione, Politecnico di Milano, 1992 and IBM-RJ 9071 (80884), November 1992.

[11] S. Ceri, J. Widom: *Deriving Production Rules for Constraint Management*, Proc. Very Large Data Bases, 1990.

[12] S. Ceri, P. Fraternali, S. Paraboschi, L. Tanca: *Automatic Derivation of Production Rules for Constraint Enforcement*, POLI int. rep. 93-21, submitted for publication (April 1993).

[13] J. Widom, R.J. Cochrane, B.G. Lindsay: *Implementing Set-Oriented Production Rules as an Extension to Starburst* Proceedings of Very Large Data Bases Conference, Barcellona, 1992.

[14] S. Ceri, G. Gottlob, L. Tanca: *Logic Programming and Databases*, Surveys in Computer Science, Springer-Verlag, 1990.

[15] S. Crespi Reghizzi et al.: *Grammatica: an open software laboratory for artificial languages*, Rept. 92-031, Dipt. Elettronica e Informazione, Politecnico di Milano, 1992.

[16] S. Crespi-Reghizzi, G. Psaila: "Federal Grammars and Modular Compilers", Politecnico di Milano, Dipt. Elettronica e Informazione, Report 93-054, July 1993.

[17] G. Psaila, S. Paraboschi: *Specification of UFI and Trigger Compilers for Chimera (Algres Testbed)*, IDEA.DE.3P.008.01, November 1993.

[18] G. Guerrini, D. Montesi, G. Rodriguez: *Specification of Run-Time Support for Chimera (Algres Testbed)* IDEA.DE.3P.007.02, November 1993.

[19] A. Sikeler, *Mapping Chimera Type and Class Definitions to Flora*, IDEA.WP.3E.019, September 1993.

[20] M. Lopez et al., Document on Technology Trasfer to ADS, April 1994.

[21] C. Draxler, "A DGC Grammar for Chimera", IDEA.DD.2P.008.01, November 1993.