SSQL, A Set Based Logic Query Language for A Distributed Software Repository

Giancarlo Succi
LII/DISA
Universita di Trento
Via Zeni 8,
I-38068 Rovereto (TN)
ITALY

Andrea Valerio, Tullio Vernazza
DIST
Universita di Genova
Via Opera Pia 13,
I-16145 Genova
ITALY

George Kovacs
CIM Research Laboratory
Computer and Automation Institute
Kende u. 13 - 17,
1111 Budapest
HUNGARY

Abstract: Nowadays, information plays a fundamental role inside every kind of activity: its ever-growing importance does not regard only computer science but also concerns a huge number of everyday activities. Information is useful only if it is structured and manageable: this is perhaps the main goal of all the information systems.

This paper describes an original alternative to the same old graphical forms used to query a repository: a set based logic query language. The query technique proposed is based on a declarative language that emphasises the set as main data type. In particular, sets are a natural and intuitive model for structured data collections.

The query language produced, called SubSet Query Language (SSQL), expands the functionalities of the starting language and introduces the capability to access an external repository. A class of new built-in predicates has been added to the starting language and the compilation and the execution environment have been adapted to the new functionalities introduced.

The SSQL paradigm has been integrated inside a distributed software artefact library and use statistics are being gathered. An initial evaluation of the SSQL paradigm, based on the experience so far acquired, shows that, in the phase of an initial effort to experience the methodology, great efficiency and flexibility can be obtained.

Giancarlo Succi received his Laurea Degree in Electrical Engineering from the Universita di Genova, in 1988, the M.Sc. degree in Computer Science from the State University of New York at Buffalo, in 1991, the Ph.D degree in Computer and Electrical Engineering from the Universita di Genova, in 1993, respectively. Since 1993 he has been Assistant Professor at the Universita di Trento. His research interests are in software engineering with an emphasis on software re-use.

Andrea Valerio received his Laurea Degree in Electrical Engineering from the Universita di Padova in 1995. Since 1995 he has been a graduate student at the Universita di Trento. His research interests are in software engineering with an emphasis on software re-use.

Tullio Vernazza is Associate Professor at the Universita di Genova. His professional interests cover software engineering with an emphasis on software re-use.

George Kovacs received the B. Sc. degree and the Ph.D degree in Electrical Engineering from the Technical University of Budapest in 1966 and 1971 respectively. He was a candidate to sciences degree (CAD/CAM in Electronics) from the Hungarian Academy of Sciences in 1978. As visiting researcher he spent one year in the USA, two years in the then Soviet Union, and one year in the Federal Republic of Germany. He also spent six months in Mexico as Visiting Professor. As founder of the CIM-EXP Development and Consulting Ltd, he has played a major role in the management of the company. His professional interests are in co-operative knowledge processing.

1. Introduction

One of the most interesting applications of computer science concerns the information systems: typical examples are business management information systems, information systems for register of birth, marriages and deaths and booking information systems.

In all of these applications there is a need for collecting and organising, possibly in an automatic way, a huge quantity of data, storing this information on appropriate memory supports.

Information, intended as the collection of data needed to carry out a specific activity, plays a fundamental role inside every kind of activity: its ever-growing importance does not regard only computer science but also concerns a huge number of everyday activities.

In particular, information is the chief concept around which an information system is built. An information system can be defined as a set of automatic tools, procedures, human resources, information flows and organisational rules and regulations aimed to manage a collection of information that an organisation or a company needs in order to pursue its aims. In particular, for the purpose of this paper, the interesting parts of the information system are those concerning computer science, i.e. automated tools and procedures to manage the information. Two are the main characteristics of these systems:

- the information and how it is structured and stored.
- the functions and processes that work on the information in order to manage the data or to produce new data from the existing information.

Concentrating on the second aspect just outlined, among the functions that should be available in an information system, the following are to be considered:

- classify and store new information
- modify or delete the information already stored
- search and retrieve the information in the system.

Regarding the first functionality introduced, it has long been studied and a lot of solutions and proposals are available in the literature. The second one is quite a design and implementation problem. The third functionality represents the context within which the work presented in this paper is placed. One of the main aspects of an information system is constituted by the search and retrieve functionalities given to the user of the system: if it is important to classify and structure the information inside the repository, as it is at once important to be able to efficiently search and retrieve the desired information.

The query mechanism is usually based on a graphic interface: a graphic form is presented to the user and he/ she has to cover some field, specify the characteristics and the bonds of the information which he/she would like to find in the repository. The result of this kind of query, most frequently, is a list, with fixed fields, of all the information present in the repository that satisfies the user specified characteristics and binds. Searches are performed following a limited set of patterns and using a standard set of initial information. Notwithstanding such methodologies is, most of but not all the time, simple and intuitive; they restrict the user to follow a specific search pattern, given a specific set of starting information. One of the biggest drawback of these graphical interfaces is the lack of flexibility, and this turns into a lack of efficiency.

A possible alternative, being impossible to realise all the forms that a user could need for his/her searches, could be to produce a fully programmable interface: quite as a programming language (SQL is a well -known example of such query languages). But the user should learn and experience such language in order to state a query. This prerequisite cannot be imposed on the user.

An interesting solution could be the use of a declarative language in order to state the queries: declarative languages are especially simple and intuitive and do not require special knowledge about computer science and programming paradigms. Moreover, a declarative language offers full computational capabilities.

Among the declarative languages developed, a language that shows as main data structure, the set could be a good choice: structured data collection is indeed naturally modelled by sets. The interface depicted allows a great degree of flexibility and efficiency, together with a full elaboration capability, without renouncing to the necessary simplicity and intuitiveness needed to deal with any kind of user.

This paper is organised as follows: Section 2 briefly outlines the characteristics of the declarative language used to implement the query technique just depicted, Section 3 describes in detail the main characteristics of the SSQL methodology and Section 4 presents some experiences with the SSQL paradigm, together with future work.

2. An Overview of SL

The distinctive idea underlying declarative programming is to let programmer work at a high level of abstraction, expressing what his/her program must do rather than how it will do it [5,6,15].

This is the reason why declarative languages present many advantages against the imperative ones:

- various kinds of static analyses are intrinsically simpler
- the program is easier to understand and its correctness, when possible, can be more simply verified
- the programs are more concise and readable, which makes declarative languages suited to fast prototyping
- parallelism is implicit, and the compiler can recognise it without great difficulty.

Furthermore, given the high level of abstraction of declarative languages, in principle a logic program can be ported on any kind of architecture, since it is up to the compiler transform it into an executable that can run fast and reliably on the desired machine.

Actually, the differences between the computational paradigms of declarative languages and the ones peculiar of the existing architecture models make an isomorphic compilation impossible: attempts to build machines dedicated to the execution of declarative languages did not have a great success.

At present, declarative languages execute mostly on standard architecture models, being these either sequential or parallel, using an executor, usually known as Abstract Machine, modelling the needed structure: this is, for example, the standard implementation technique for Prolog, which uses the WAM (Warren Abstract Machine) as an intermediary between the compiler and the real machine.

The choice of sets as the basic structure of a programming language comes from the general conviction that they are a very powerful programming tool, particularly suited for fast prototyping, where it is useful to benefit high-level data structure and operations.

Set theory can be used to represent problems in various areas of application.

Moreover, sets are a natural model for representing structured data collections, and in particular collections of organised information items.

Sets have been introduced in many programming languages, yet only a few embody them as primitive objects, providing some basic operations as part of the language.

Other declarative languages, like Prolog [9,16], have added to their capabilities that of handling sets, but with not very satisfactory results: in Prolog, for instance, sets are represented as lists, imposing an arbitrary order to their elements, so they are managed as ordinary terms and the built-in predicates that support them are not fairly efficient.

In the late 70ies a group of researchers at the New York University designed a set based language: SETL [22].

SETL allowed the programmer to define the set type that better met the problem specifications, thus adopting an automatic strategy to single out the best representation.

It has demonstrated how sets can be expressive and how many applications using them may be thought of.

This work is based on a declarative language, called SL (Set Language): SL is designed around sets, handling them in a clear and simple way and offering the basic operations on them as part of the language [18].

SL can be viewed as a superset of SEL (Subset Equational Language), a logic-functional language developed in 1987 by Jayaraman and Plaisted [2,3,4,10].

SEL joined to equational programming, whose paradigm was already consolidated, the idea of introducing subset assertions devoted to procedures that have sets as result.

SL adds to programming paradigms of SEL the standard notations of set theory, allowing the user to easily write programs dealing with sets.

SL is based on equational assertions and uses the following kinds of constructs to deal with sets:

- the first construct of SL allows to build a set with the typical set notation, as in:

```
cartesian_product(S,Z) = { (X,Y)
: X in S, Y in Z }.
f(S) = { h(X) : X in S; X < 3 }.
```

  The former produces the Cartesian product of two input sets, whereas the latter returns a set of functors h/1 each having as argument an element of the given set S which respects the condition to be lower than 3.

- assertions like the following are used to build a set whose elements are a sequence of integers, in case with a given step:

```
f(X,Y) = {X..Y}.
g(X,Y,Z) = {X,Z..p(Y)}.
```

  In the latter Z is the second element of the sequence: the step is therefore given by Z-X.

- pattern matching can be applied on set elements, as in the following assertion:

Phase	Purpose	Input	Output
precompilation	translating source SL code to SEL code	SL code	SEL code
	translating SEL code to SAL code	SEL code	SAL code
optimisation	optimise SAL code	SAL code	optimised SAL code
codification	translating SAL code to SAM numeric code	optimised SAL code	SAM numeric code

```
 aFatherTeacher({father(X,_)|_},
{teacher(X)|_}) = X.
```

This assertion identifies a person having children and being a teacher.

- the operator union is used in assertions like:

```
f(X)=g(X)union h(X).
```

where f, g and h return a set as result.

The implementation of SL takes place in two phases:

1. the development of a compiler targeted to an abstract machine
2. the implementation of an abstract machine on the real architecture.

The compilation and execution of SL programs are analogous to those of Prolog: adopting an abstract machine allows to get free from the constraints of the real architecture on which the program will be executed.

Portability of SL programs is therefore due to the implementation of the abstract machine: the compiler transforms the source code into a sequence of assembly instructions of the abstract machine independently of the way how the instruction set is implemented.

The compiler can thus devote its attention to efficiency and reliability of the executable.

The SL abstract machine is called SAM (Set Abstract Machine) and its assembly language is obviously called SAL (SAM Assembly Language) [12,20].

The SAM belongs to the WAM family since its general structure resembles quite a lot that of the WAM [1,13].

However it does not require full unification capabilities, therefore there is no need of the trail.

The SAM can be viewed as a sister of the SEL-WAM [7], from whom it inherits most of the implementation strategies, which are extended by some new optimization techniques, table of constants and the capability of handling functors.

The process of compilation is articulated into various phases: each of them aims to pass from a higher to a lower level code, step-by-step solving problems that arise while passing from the declarative paradigms of SL to the imperative ones of SAM [8,11,17]. Compilation phases are summarised in the table above.

SL is designed for set theory and therefore it lacks explicit control, as it is evident in the following SL assertions:

```
makeUnion(Set1,Set2)=h(Set1)unio
n g(Set1,Set2).
prime(Numbers)={X:X in Numbers;
empty(divisors(X,Numbers{X}))
divisors(X,S)={Y:Y in S; Y <> 1
, (X/Y)*Y == X }
```

The first phase of compilation consists in translating SL to SEL: a precompiler transforms SL constructs into sequences of equational or subset SEL assertions.

The SEL code for the above-mentioned SL assertions is:

```
makeUnion(X, _) contains h(X).
makeUnion(X,Y) contains g(X,Y).
prime({X|T})contains=
if(empty(divisors(X,T)))
then {X} else {}.
divisors(X,{Y|T})  contains  if
((Y<>1) && ((X/Y)*Y==X))
then {Y} else {}.
```

The second example shows a remarkable feature of SL, viz. the multiple matching: since set elements are not ordered, a matching of the kind X in Numbers produces the matching of X with all the elements of the argument set.

Therefore, the resulting set contains all the elements of the given set, that satisfy the condition empty(divisors(X,T)).

SEL code produced by the precompiler is then processed by the compiling module, whose task is to yield a SAL code, implicitly optimized in order to be efficiently and reliably executed on different kinds of architectures.

The focal point of the compilation is in the translation of assertions dealing with sets.

There are two kinds of operation that an assertion can perform on a set:

- mapping one set onto another;
- searching for particular elements of a set.

The execution of machine instructions devoted to these operations is onerous in terms of machine resources: hence it is important that the use of such instructions should be optimized.

Moreover, identifying a set element imposes to check the correctness of the search and to build a code that permits to retract a wrong choice, when the failure of a pattern matching depends on the previous matching of another pattern.

Steps allowing to perform such a task make up the SAL Code Implicit Optimization Algorithm.

Implicit SAL Code Optimization Algorithm aims to produce a sequence of SAL instructions that ensure correctness and completeness of the result, ordering all the patterns to be matched in a set, such that to minimize the possibility of failure in matching and to reduce the search space when retracting a wrong choice.

The SAL code produced by the compiler is the input of an optimizing module, which applies on it the following optimization strategies:

- ORA (Optimized Register Allocation): it consists in trying to use the lowest number of registers, avoiding unnecessary data movements, and to re-use registers whenever possible.
- RIE (Redundant Instructions Elimination): it sometimes happens that one or more instructions are correct but redundant and can thus be eliminated.
- ET (Environment Trimming): it consists in ordering the permanent variables on the environment of the assertion as well as in reflecting the ordering of their last occurrence on the right side of an assertion. Namely, the variable whose last occurrence is the last of all will be the first on the environment, and so on. This allows to deallocate parts of the environment as they are no longer used, saving dynamic space allocation.
- LCO (Last Call Optimization): it is based on the fact that permanent variables allocated to an assertion should no longer be needed after having prepared all the arguments for the last assertion call on the right side with the put instructions. Then, the environment can be deallocated right before the last assertion call.

The translation from SL to optimized executable SAL is performed by a Prolog written compiler.

3. The Set Based Logic Query Language SSQL

The overview of the SL language presented in Section 2 has allowed to outline its potentialities, in particular concerning the management of information that can be well represented by sets.

The subset assertions of SL are a powerful mechanism that permits to deal with repositories containing structured information, for example the electronic repository of a library. In this sort of a database there are classified all (or at least many) of the books that can be found in the library. Each book is catalogued using a set of information, such as author, title, editor and so on, and an unambiguous identifier (for example the Dewey code) that characterises univocally every single book.

This electronic card, or information unit to be referred later on, and more generally the electronic repository of a library, provide only one possible example of structured repositories where the methodology presented in this paper can be successfully applied.

Usually the searches into these repositories are made by appropriate systems: these systems present to the user an interface based on a form that has to be filled by the user. These forms are fixed and allow settled searches, based on one piece of (or, sometimes, more) information that the desired element must have (what is called a bond). Often the result of these queries is rather unsatisfactory and not focused, and it can be improved by only repeating and detailing the query.

In this context there dawned the idea of realising a query methodology that should be flexible, simple and intuitive, and allow the user to perform searches in a more precise way.

The idea is to build a flexible and efficient interface between a user and a database using a declarative language as the underlying methodology.

The decision of using SL in the development of SSQL is due to the sophisticated and intuitive management of sets that it performs: the information contained in the repository can be well represented by a set made up of structured elements, that represent the information units stored in the database.

The power provided by this methodology is manifold:

- simplicity of the language: SL is particularly intuitive and simple to use and it does not require much effort to learn it. Queries can be built immediately following simple models that are given.
- search flexibility: the possibilities offered by SSQL are theoretically countless. It is possible to formulate queries based on data that are really owned by the user, without built-in ties on their structure.
- search efficiency: results of the searches are particularly targeted and satisfactory, thanks to the possibility to customise the queries, carefully specifying the bonds.
- full elaboration possibility: SSQL is based on the SL language and allows to elaborate the data gathered via queries. It has not to be forgotten that SL is a declarative programming language, complete and effective, particularly concerning problems where complex structures must be managed using sets as models for computations.

The library context where we have fitted up SSQL is only one of the possible infinite situations where it could be useful to adopt a declarative query language as SSQL is.

Using the SL language in order to query a repository means to expand the language itself with a class of new suitable predicates. SL does not provide access to an external data type: the new capability that has to be introduced is the possibility of managing a particular data structure, or in other words, of accessing an external database.

The first step that has to be taken is to create inside SL a paradigm that allows to access an external database, opportunely identified, yielding the acquisition and the elaboration of data contained into the archive feasible.

Besides, among the paradigms provided by SL, a class of specific assertions has to be inserted aimed to evaluate the information associated with each single unit contained in the repository. This aspect heavily affects the efficiency and performance of the whole system: in order to search the contents of a database, it is possible to query for each single piece of information, but this is really time-expensive, or it is possible to store part of the repository in the environment memory space, but this choice is too much space-consuming. The solution that has been used is half-way these two alternatives: an unambiguous identifier is used as the primary key to retrieve an information unit in temporary structures inside the random access memory, while following the operations concerning the same information unit that take place inside the memory.

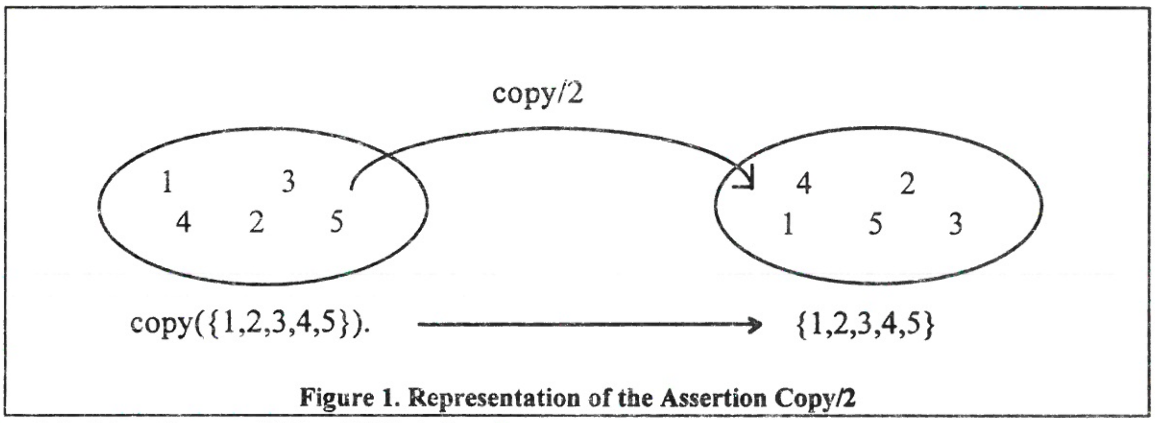

Figure 1. Representation of the Assertion Copy/2

This cache mechanism improves the efficiency of the class of predicates that realise the evaluation of data contained inside the information unit cached in the memory and, together with the predicate that identifies an external database, constitutes the core of the SSQL language.

All these predicates have to be translated, during the compilation phase, into an appropriate sequence of imperative instructions that, when executed by the abstract machine, will realise the access to the external repository.

The extension of the SL language with all the new functionalities just presented for the management of an external repository has been named SSQL, an acronym for SubSet Query Language, a language to query a database using set and subset constructs.

Following there are described in detail the extensions and the new predicates introduced into the SL language, and the changes made on the compilation process and, in the end, the work done to adapt the abstract machine to the new functionalities provided.

3.1. The Syntax of SSQL

SL provides various data structures, among which there are sets, lists and strings, but it does not supply adequate constructs to manage information coming from an external data structure.

It is necessary to expand the characteristics of the language with this new functionality: particularly, it has been decided to modify the instruction that SL uses in order to build a set.

Starting from the equational assertion:

```
copy(X) = {Y: Y in X}.
```

which simply creates a new set made up of the elements of the set X passed as argument (see Figure 1), the idea is to extend this assertion in order to build a set with the elements contained in an external data structure.

SL could be provided with an assertion like the following one:

```
copy(repository)= {Y : Y
in repository}
```

whose meaning is: build a new set using the elements contained in an archives identified by the keyword repository, re-writing the term copy (repository) in {Y: Y in repository} as shown in Figure 2. The set built with this assertion, and stored as a regular structure by SL, contains only the unambiguous identifier of each information element present in the repository: this unambiguous identifier can be viewed as an alphanumerical string that plays the role of a primary key in the database.

The assertion , just shown, specifies among the arguments the keyword repository, but this syntax is not efficient and upgraded. The choice has been to use an implicit notation, requiring that the keyword is specified only inside the body of the assertion, thus correctly identifying the source of information.

A step further, it has to provide the capability of testing the data joined to each single information unit contained in the repository. It could happen,on minimising the database access, to store all the data present in the repository in the random access memory, then using the standard predicates of SL to evaluate and elaborate the information. This solution is plainly unfeasible because of the obvious memory space that would be required. The solution adopted was to store in the SL environment only the identifiers of the information unit, retrieving and caching each information unit only when expressly needed during the elaboration, then discarding all the data no more useful. SL has been provided with a series of built-in assertions that allow to test a specified attribute of an information unit marked out by the identifier passed as argument to the assertion.

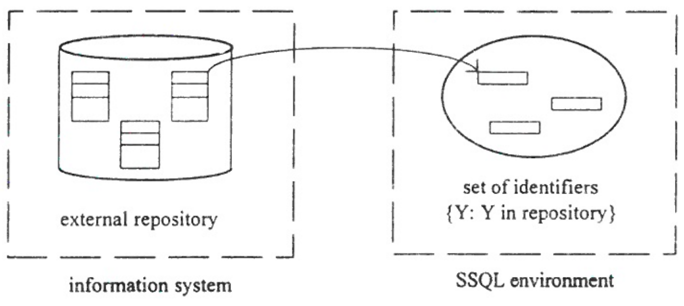

Figure 2. Representation of the Access to An External Repository

An example would explain this technique: the search, inside a library, of a book written by 'Asimov' can be done calling the assertion:

```
searchWithAuthor(Author)          =
{Book:    Book    in    repository;
author(Book)= =Author}.
```

through the query:

```
searchWithAuthor( 'Asimov' ).
```

The main operations underlying the execution of this assertion are intuitive ones (once more demonstrating the simplicity of the SSQL paradigm): the identifiers of each catalogued book are taken from the repository and for each book the relative information is evaluated in order to test if the author of the book is 'Asimov'.

The SL language has been enriched with special assertions that receive as input argument the unambiguous identifier of an information unit and, after retrieving the desired data from the database, return a specific attribute, so that it is possible to compare its value with a given value.

These new assertions can be divided into two classes:

- *specific assertions*: these assertions refer to a precise attribute of the information unit that has to be evaluated. The syntax of these assertions is:

```
value = preciseAttributeName(identifier)
```

where value is the value of the precise attribute managed by the assertion and associated with the information unit characterised by the identifier. The keyword preciseAttributeName characterises the specific attribute that must be evaluated and depends on the structure of the information unit stored in the repository. In the library environment, the attribute that has been implemented is:

```
author, location, type, title,
date, editor
```

If the information unit checked by one of these assertions has not the attribute specified by the assertion, the result is an empty set.

- *generic assertion*: this kind of assertion allows to specify as input arguments the name of the attribute that has to be evaluated, together with the identifier of an information unit, and returns the value of the attribute. The syntax is:

```
value=test(identifier,attributeName)
```

where value is the value of the attribute attributeName associated with the information unit characterised by the identifier. This assertion is characterised by the keyword test. In this construct the user must specify the name of the attribute to test: this permits high flexibility but also could cause various errors i the name of the attribute or the kind of data associated with it .

The extensions to the SL language that have been made, together with the new assertions , allow to manage an external repository from inside the SL environment. One more thing lacks: how is the external database identified and located? For this purpose a precise instruction has been provided which associates with the keyword repository an identifier locating the desired database. The syntax of this instruction is the following:

```
createExternalLink =
(repository, identifier).
```

The actions underlying this instruction are to link the keyword repository to the specified identifier, storing this identifier into a suitable memory location. The identifier will be retrieved and used in the construction of the data structure that will act as a reference to the external repository.

An example can summarise this introduction to the new functionalities provided by SSQL. The search for the location of a book of 'Asimov' titled 'The Foundation', in particular an edition that has some pictures, can be performed using the assertion:

```
searchWithAuthorTitlePictures(Author
, Title) = {[Book, Location]: Book
in     repository;     Location    is
location(Book);  author(Book)  =  =
Author,   title(Book)   =  =   Title,
test(Book, hasPictures)= = 'yes').
```

The query:

```
searchWithAuthorTitlePictures
( 'Asimov', 'The Foundation').
```

will result in a set containing couples of elements: the first is the identifiers of the book, while the second gives the location where the book can be found. If there is no one book that satisfies the query, a set containing a couple with empty elements will be returned.

3.2. The Compilation of SSQL Programs

The compilation process of an SSQL assertion is carried out in three steps:

1. precompilation: the aim of this phase is to translate the SSQL construct, based on the construction of a set with a classic mathematical notation, into an assertion that is comprehensible to the compiler. This task is accomplished by an apposite module, the precompiler, which takes as input the SSQL code and verifies the correctness of its syntax, also performing a first semantic check.
2. compilation: after the precompilation step, the assertions are passed to the compiler module, which executes the translation from the logical query language SSQL to the imperative language SAL
3. linking: this phase allows for linking more object SAL modules together into an executable module and for linking this module with the object code obtained from the query, thus obtaining the executable query module.

Each one of these phases is detailed in the next paragraphs.

3.2.1. The Precompilation Phase

The function of this phase is to check the syntax of SSQL programs and then to re-write them using equational and subset assertions.

Starting from the syntax check of a SSQL assertion, in particular of the body of that assertion, the model checked by the precompiler follows the pattern:

```
...   =   {predicate:   associations;
definitions; conditions}.
```

The analysis of the predicate aims to check whether it observes the SL grammar. All the variables found during the scanning process are included in an appropriate list.

Then the precompiler tests the presence of the symbol : and the following associations are evaluated. Each association is checked against the model:

```
        variable in argument
```

Two actions are carried out:

- the variable that has been found is searched in the list produced during the previous step and if not present on the list, it is included in a different list of variables that have to be verified during the scan of the definitions.
- the argument found is compared with the arguments specified as parameters in the head of the assertion and, if not among them, a syntax error occurs. At this step the precompiler is instructed to identify the keyword repository when it is found as argument: when the keyword repository is found, the precompiler enters a particular state, purposely designed to handle the SSQL assertions.

Next step aims to evaluate the definitions: if in the previous step the keyword has not been found and the precompiler is not in the right state to manage a SSQL construct, an error occurs. A definition follows the pattern:

```
        variable is expression
```

Each variable is checked using the list created during the associations analysis: if a variable used inside a built-in SSQL predicate, such as author or location, joins the keyword repository, then the process goes on with the substitution of the unbound variable inside the predicate with the expression specified in the definition. Otherwise, an error occurs.

On ending the analysis of the assertion's body, the precompiler verifies if there are conditions, i.e. if there are predicates that have to be satisfied with the terms identified by the specified associations. Here are two examples of feasible conditions :

```
...; X > -1, X < 1, Y = 0 }.
...; f(X,Y), g(X) = = Y }.
```

where X and Y are unbound variables used in the predicate, f/2 is a functor and g/2 is an assertion, while other symbols have the usual meaning. The specific predicates of SSQL are recognised by the precompiler and if the keyword repository in the previous steps has not been found , an error occurs, otherwise their syntax is verified.

After completing these steps, when the SSQL assertion gets correct, the assertion is re-written with an equational and subset assertion:

- a new equational assertion is created with the head of the SSQL source assertion and as body of a call to an assertion with name extSetMaker___n (where n is a numerical parameter). This new assertion aims to explicitly pass the input arguments and to identify the pattern of each argument
- a new subset assertion is created with name extSetMaker___n and whose arguments render explicit the pattern of the elements that must be bound to the unbound variable which was found during the analysis of the predicate (replaced by the appropriate definitions when necessary). The body of the new assertion will have the structure:

```
... contains if ( conditions )
then { predicate } else {}.
```

The variable associated with the external data that must be elaborated is replaced by the keyword ExtElemId and its resolution will be remitted to the compiler.

An example can help understand the precompilation process. Given the SSQL assertion:

```
searchWithTitle(Title)      =      {
[Book,  Author,  Location]:  Book
in repository; Author is author
(Book),   Location   is   location
(Book); title(Book) = = Title ).
```

after the precompilation the following two assertions will be produced:

```
        searchWithTitle(Title)=
        extSetMaker___0(Title).
```

```
extSetMaker___0 (Title) contains
if ( title(ExtElemId) = = Title)
then {[Book, author (ExtElemId),
location (ExtElemId)]} else {}.
```

3.2.2. The Compilation Phase

After the precompilation phase, equational and subset assertions have to be elaborated by the compiler, that translates them from the declarative language into the imperative language of the abstract machine.

This section presents the details concerning the compilation of the assertions which allow the access to external data, showing, in particular, the imperative instructions thereby which a SSQL assertion is translated.

The compilation process starts with syntactic analysis, lexical analysis and semantic analysis. These analyses follow the usual principles used to compile a generic SL assertion, but also introduce a characteristic aspect: the scanning of a subset assertion that includes access to the external repository would produce an error (according to the usual SL patterns) by identifying the literal 'ExtElemId' as an unbound variable.

The scanning patterns of the compiler must be extended by introducing appropriate schemes for the compilation of SSQL assertions.

The compilation of an equational assertion made when access to external data is requested is fully compatible with a standard SL equational problem, and its translation is performed without problems by means of the SL compiler diagrams.

On the other hand, for a subset assertion produced when access to external data is requested, the SL compiler has to be extended by a set of rules that allow to produce the appropriate imperative instructions.

The first step that the compiler must take is to correctly identify these particular subset assertions: this step is rightly performed by the SL compiler, that makes the assertions join the usual subset construct:

```
        try_sub_and n
            ...
        proceed_sub
```

Next step is the usual analysis of the arguments in the head of the assertion, but a particular operation has to be run: a reference to the external data structure used must be generated. A subset assertion produced by the precompilation of a SSQL assertion, has a name that begins with extSetMaker___ and when, during the compilation, this string is identified then a reference to the external data structure has to be built. The reference is built using a set of rules that:

- create a reference to the external data and store the reference into the memory structure of the abstract machine.
- store in a register the pointer to the structure just created.

The reference to the external data contains two pieces of information:

- an identifier that allows the abstract machine to identify the external data structure at running time.
- a pointer to the specific information unit contained in the repository, that must be accessed.

The reference structure created is depicted in **Error! Reference source not found.**: the label EXT_DATA characterises the structure as a reference to external data. The identifier of the repository that follows the label is the second parameter of the assertion createExternalLink.

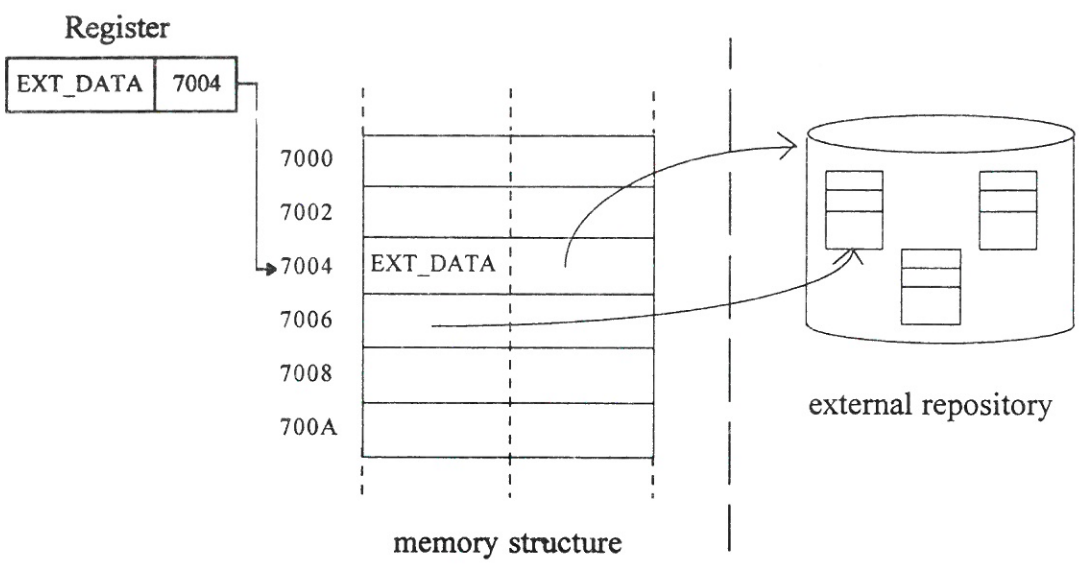

Figure 3. Reference Structure to an External Repository

The SAL instruction used to create the reference to an external data structure is external_link Zi, which does all the operations just described and puts in the register Zi a pointer to the structure created.

When the head of the subset assertion has been analysed, the compilation goes on examining the body of the assertion.

Generally, the operations that have to be done are the identification and the elaboration of a specific element inside the external data structure and its copying , if necessary only after one or more conditions specified in the query have been positively verified, in an appropriate set. These operations must be accomplished for each element of the external data structure, resulting in the creation of a set of identifiers to be stored in the memory structure of the abstract machine.

The instruction used to translate the body of the assertion is the map construct.

Error! Reference source not found. summarises these operations.
Disregarding possible condition or elaboration that has to be executed on the elements, the acquisition of the identifier of the elements stored in the repository is made in the following steps:

1. the instruction map_over Za Zi Zm end, where Za contains the pointer of the reference created with the instruction external_link Za, reads the identifier of the external data structure and the value of the identifier of a specific element inside the repository (that has been set to zero with the instruction external_link) contained in the reference pointed by Za, and accesses the repository by retrieving the identifier of the first element. This identifier is stored in a temporary memory location and the address of the memory location is put in the second field of the register Zm, while in the first field of Zm there is stored the label EXT_ELEM.
2. the instruction insert Zo Zm is used to verify the label in the first field of the register Zm: if it corresponds to EXT_ELEM then the identifier stored in the temporary memory location is moved on a new location inside the constants table, the address contained in the second field of Zm is adjusted and the element pointed by Zm is inserted into the set identified by Zo.
3. the instruction end_map_over Zi Zm starts access to the external data structure by retrieving the identifier of the next element. If all the elements have been retrieved then the execution jumps to the instruction following the end_map_over, otherwise the identifier of the new element is stored in the temporary memory location and the value of the second field of Zm and of the pointer to the current element in the reference

structure is adjusted. The execution then returns to step 2.

The presence of possible conditions in the initial query determines, during compilation, the insertion of a conditional structure that branches the execution flow depending on the result of the conditions.

Within the structure just described, that is used to translate the body of a subset assertion, a new structure is inserted, following the model:

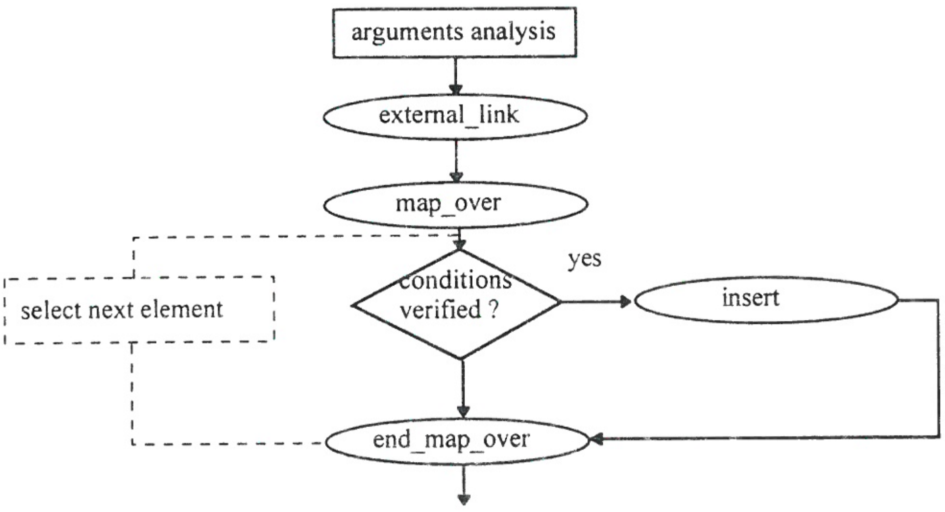

Figure 4. Instructions Used in Order To Access and Elaborate External Data

```
prepare the parameters that have
to be passed to
      the built-in predicate

operation
operatorName/argumentsNumber

prepare the parameters that have
to be passed to

      the equality check
operation ==/3
...
fjump Zi label
insert Za Zb
label: end_fjump Zi
```

This structure also shows the technique adopted in order to compile a SSQL built-in predicate: the instruction operation operatorName /argumentsNumber allows to give directions to the abstract machine for executing a built-in function (without an explicit declaration of this function inside the SSQL source code). This technique is also used when a built-in operator is compiled, such as, for example, the sum operator (+/3) or the equality operator (==/3). Before executing the instruction operation, the arguments have to be loaded in the argument register Ai and the register that will contain the result of the operations has to be prepared.

The so far described compilation process is recapitulated in the next applicative example. Given the SSQL assertion:

```
searchWithAuthorEditor(Author,
Editor) =
{Book: Book in repository;
author(Book)==Author,
editor(Book)==Editor}.
```

after the precompilation there will be produced the two assertions:

```
searchWithAuthorEditor(Author,
Editor) =
extSetMaker___0(Author, Editor).

extSetMaker___0(Author,   Editor)
contains if
(author(ExtElemId)==Author) &&
editor(ExtElemId)==Editor)   then
{ExtElemId}
else {}.
```

The SAL code produced after the compilation of these two assertions will be:

searchWithAuthorEditor_3_0:

```
   [1] try_eq_else -1
   [2] get_variable X3 A1
   [3] get_variable X2 A2
   [4] get_variable X1 A3
   [5] put_value X3 A1
   [6] put_value X2 A2
   [7] put_value X1 A3
   [8] execute extSetMaker___0/3
extSetMaker_3_0:
   [9] try_sub_and -1
  [10] allocate 6
  [11] external_link X8
  [12] get_variable Y6 A1
  [13] get_variable Y5 A2
  [14] get_variable Y4 A3
  [15] map_over X8 Y3 Y2 S_END_0
S_START_0:
  [16]   put_value Y2 A1
  [17]   put_variable Y1 A2
  [18]   operation author/2
  [19]   put_value Y1 A1
  [20]   put_value Y6 A2
  [21]   put_variable X6 A3
  [22]   operation = =/3
  [23]   put_value Y2 A1
  [24]   put_variable X7 A2
  [25]   operation editor/2
  [26]   put_value X7 A1
  [27]   put_value Y5 A2
  [28]   put_variable X5 A3
  [29]   operation ==/3
  [30]   put_value X6 A1
  [31]   put_value X5 A2
  [32]   put_variable X4 A3
  [33]   operation &&/3
  [34]   fjump X4 J_0
  [35]   insert Y4 Y2 J_0:
  [36]   end_fjump X4
   [37]   end_map_over   Y3   Y2
S_START_0
S_END_0:
  [38] deallocate
  [39] proceed_sub
```

Lines [1] - [8] contain the SAL code associated with the equational assertion searchWithAuthorEditor/3 and they do not require an explicit explanation. The SAL code associated with the assertion extSetMaker___0 begins on line [9]. Line [11] builds the reference structure to the external repository, using the identifier given by a previous instruction createExternalLink and copying the address of this structure in register X8. Line [15] begins to scan the external repository: the instruction map_over retrieves the identifier of the first element contained in the repository, storing it in a temporary location and copying the address of this temporary location in register Y2. Line [18] calls the built-in function author/2: the result of this function, i.e. the author of the element given as input argument, is compared on line [22] with the value specified in the query and the result is saved in register X6. Line [25] retrieves the editor of the external element and compares it with the editor specified in the query and the result is copied in register X5. Line [33] computes the logical AND between X6 and X5: if the result is true, i.e. the author and the editor of the current element correspond to the values specified in the query, then execution proceeds with the insertion of the identifier of the current element in the result set (line [35]), otherwise execution jumps to line [36]. Line [37] retrieves from the external repository the identifier of the next element, storing its address in register Y2 . If the current element is the last one then execution proceeds to line [38], otherwise execution jumps to line [16] and the current element is elaborated.

3.2.3. The Linking Phase

The compilation process described in the previous section represents the core of the translation from the SSQL logic language to the SAL imperative language. In the object code the function calls uses an address table that links each identifier and arity to the address of a sequence of instructions which performs the desired function.

In order to obtain an executable module from an object module, the references contained in the table must be replaced by the appropriate addresses in the source code.

The SSQL environment allows the user to create and use more than one source program simultaneously, splitting the source SSQL assertion into different modules that the system compiles in separate object modules. When an executable module must be produced using more than one object module, the operation just outlined is not feasible as presented: the function's tables and the constant's tables of each object module, including the object module obtained by compiling the query, must be brought together.

These operations are performed by a module called linker. The linking process can be divided into three steps:

- analysis of the constant's tables and of the function's tables. The constant's tables are combined into a single table and a warning message is produced if the same constant is differently defined in two different tables
- production of the executable module, using the object code obtained through linking the

object module produced by the compilation process

- production of the executable module associated with the query, using the information gathered at first step.

The output of the linking process is made up of two executable modules, one of which is obtained from the SSQL programs, and the other is associated with the SSQL query. These two modules are then passed to the abstract machine for execution.

3.3. The Execution of SSQL Programs

The abstract machine that executes the SAL code obtained through the compilation of the SSQL programs is called SAM, an acronym for SL Abstract Machine. The structure of the SAM follows the same pattern as the WAM's [7], but its functionalities have been expanded in order to manage the new SSQL constructs: this section aims at explaining some particular aspects of the SAM connected with the execution of SSQL assertions.

The execution of a SSQL query begins with the loading of the SAL code associated with the SSQL query and the programs: the loading phase and the preparation of the constants table, entrusted to appropriate set-up procedures, aim to copy the SAL code and other needed data into the memory structures of the SAM, thus preparing the environment for the execution.

The elaboration of the SAL source code is performed by an apposite module, called executor, which uses the information stored in the memory structures of the SAM, and executes the SAL source instructions, managing partial and final results. The execution of the SAL source code is done by the executor, which reads each SAL instruction and accomplishes all the basic operations required.

The core of the execution process can be condensed with a complex selection operation. From the code area there is extracted the SAL instruction, together with its arguments, of which address corresponds to that stored in the PC register. The instruction is compared with a set of mutually exclusive patterns (all of the allowed SAL constructs) and, when the right matching is found, the corresponding basic operations are performed. Then the next SAL instruction, pointed by the PC register, is extracted and the process goes on. The basic operations associated with each SAL construct depend on the data type contained or pointed by the arguments of the instruction considered a detailed description of the SAL instructions set and the associated basic operations can be found in [14,18].

In order to recognise and properly execute a SSQL assertion, the SAL instructions set and the basic operations associated with the same SAL instructions have to be modified.

The external_link instruction, being a new SAL instruction introduced with the SSQL paradigm, requires the modification of the SAM with the introduction of a new instruction type. The basic operations that must be performed when the instruction external_link Za is encountered are the following:

1. in the first empty location of the heap, pointed by the HP register, here is created the reference structure to the external repository and in the first field of the structure the EXT_DATA label is stored
2. the address of the identifier associated with the external repository of the SSQL instruction createExternalLink is saved in the first field of the reference structure just created
3. the first field of the next location in the reference structure is set to zero, indicating that no previous access to the repository has been made
4. the address of the structure created with the previous operations is stored in the register Za.

These operations create the reference structure to the external repository that will be searched during the execution of the SSQL query.

The instruction

```
createExternalLink              =
(repository, identifier).
```

is resolved during the compilation phase and does not require any modification of the SAM. During the compilation of the SSQL programs, when this instruction is found, the identifier associated with the external repository is stored in an appropriate memory location. The identifier is then transferred to the SAM the same way as the constants contained in the constants table are.

The abstract machine already has the right patterns to deal with the other SAL instruction used to manage the information coming from the repository, i.e. the map and insert constructs, but it is not able to properly deal with the new external data types. The basic operations that are executed when these instructions are encountered must be adapted and specified in order to deal with the new external data types.

Beginning with the instruction map_over Za Zi Zm end, when it is encountered the first operation that must be performed is to verify the label contained in the first field of the register Za. If the label corresponds to EXT_DATA then the execution flow is directed to the specific instructions sequence that deals with an external data type, otherwise the standard procedure is accomplished.

The operations performed in the case that the label corresponds to EXT_DATA are:

1. the external repository retrieve procedure is called with input arguments by the identifier of the repository, retrieved from the reference structure addressed by the register Za, and the pointer to the current element inside the repository, obtaining the identifier of the next element contained in the repository
2. if the value retrieved at step 1, i.e. the identifier of an element contained in the repository, is zero then the execution goes on with the instruction associated with the label end. Otherwise, if the identifier is not zero, it is saved in a temporary memory location at the beginning of the constants table and its address is copied in the second field of the register Zm, while the EXT_ELEM label is written in the first field of the register Zm
3. the address of the identifier associated with the element retrieved, is saved in the reference structure, so that the pointer should be updated to the current element
4. the current state of the elaboration is passed to the instruction end_map_over copying the value of the register Za in the register Zi.

The instruction end_map_over Zi Zm start is treated the same way as the map_over is, and the operations that must be done are:

1. the external repository retrieve procedure is called by input arguments the identifier of the repository, retrieved from the reference structure addressed by the register Zi, and the pointer to the current element inside the repository, resulting in the identifier of the next element contained in the repository
2. if the value retrieved in the previous step is zero then the current element was the last element contained in the repository and the execution goes on with the instruction following the end_map_over. Otherwise, if the identifier is not zero, it is saved in a temporary memory location at the beginning of the constants' table and its address is copied in the second field of the register Zm
3. the address of the identifier, associated with the element retrieved, is saved in the reference structure, so that the pointer should be updated to the current element, and then the execution flow jumps to the instruction addressed at the label start.

The instruction insert Zo Zm is used to manage the external data, in particular in order to insert the identifier of an element in the results set. The technique adopted to identify when external data are used is to check if the label contained in the first field of the register Zm corresponds to EXT_ELEM. When the label is EXT_ELEM then the sequence of operations that has to be performed is the same as that required for managing a standard SL data type.

The only exception is that the current element, i.e. the identifier associated with an external element, must be copied in the constants' table, while updating the label in EXT_ELEM to LIT and updating the address of the identifier.

The instructions used to prepare the arguments before the call of a built-in operator, i.e. the put instructions, do not raise problems. The required operation is simply to copy a value in an argument register and the abstract machine is able to deal with reference to external elements.

The built-in SSQL assertions allow to deal with the attributes associated with the elements contained in the external repository. These assertions are compiled using the SAL instruction operation, which specifies the name of the built-in operator used, followed by its arity. During the execution, the abstract machine must recognise these operators and call the function specified as argument. The identification of the required operator is performed through comparing it with the set of the built-in operator's patterns, selecting the right one in a way similar to the identification of a SAL instruction during the execution. The built-in attributes are executed by a specialised function, which performs all the computations needed and also calls an external repository retrieve procedure when necessary.

3.3.1. The Access To the External Repository

Access to the external repository is performed through an appropriate interface that encapsulates the internal structure and the information managing mechanisms of the repository. The encapsulation technique adopted allows to use different repositories if only modifying the low level code associated with the basic operations performed by the methods of the interface, without modifications in the syntax of the methods themselves and in the SAM basic operations that call them.

Two main methods of access to the external repository have been built:

1. the method get_next(DBId, elemId) returns the identifier of the element, contained in the repository, next to that specified by the identifier elemId, where DBId is the identifier associated with the external repository. If the value of the argument elemId is zero, as for instance when a map instruction is executed, then the identifier of the first element contained in the repository is returned
2. the method get_ext_data(DBId, elemId, attributeName) requires as arguments the identifier associated with the external repository, DBId, the identifier of the current element, elemId, and the name of the attribute that has to be retrieved, attributeName. This method returns the value of the attribute attributeName of the element elemId contained in the repository DBId, or, if the specified attribute is not defined for the current element, a value equal to zero.

4. Conclusions

This paper describes the implementation of a set based logic query language, called SubSet Query Language (SSQL), an interesting and promising new technique to search for and retrieve information in a generic repository.

Usually, searches inside a repository are performed through fixed graphical forms that must be filled in by the user: this methodology, of course simple and friendly, sets some limits to the typology of possible searches that can be done, restricting the user's queries to a finite set of fixed forms.

The work presented in this article proposes a search technique based on a logic language, that combines simplicity and friendliness with search flexibility and efficiency.

Queries are stated using the typical constructs of a logic query, in particular a paradigm that emphasises as main data type the set, expanded in an appropriate way, so that to include the predicates that allow that the information contained in an external repository is accessed.

The intuitiveness and simpleness of the logic paradigms, together with their characteristic that only the logic structure of a problem must be expressed, render logic languages as the most suitable candidates in order to realise this new query technique.

The implementation of this technique has been based on the set based logic language Set Language (SL), introducing a class of new predicates that extend the functionalities of the language, allowing access to an external repository. The SL environment, i.e. the compiler and the abstract machine, has been modified in order to manage the new predicates and the information coming from the repository.

The SSQL logic query language tool that has been built, has been integrated into a distributed software artefact library in order to gather use statistics and information.

The verification on the field of the SSQL methodology, concerning an objective evaluation (i.e. the time needed to return the results of a query) and the efficiency of searches (i.e. the satisfaction level of the user expectations versus the effort spent to state the query), is still incomplete and has not allowed to outline a final evaluation.

The results that can be extrapolated from these initial experience have emphasised some particular aspects: the first approaches of the user to this technique has shown some difficulties, mainly due to the initial effort needed to get familiar with SSQL. In particular, while the use of a graphical form does not request any mental effort, on the other hand writing a SSQL query requires to ponder on and to organise the problem that must be solved. This initial effort is, at the beginning, a drawback, but, when a little experience has been acquired, it allows to obtain rewarding query results. The initial effort is recompensed with great search efficiency and search results that fully meet the user expectations.

It must not be forgotten that SSQL is based on a set based declarative language with the same computational properties as other imperative language, for example C or Pascal, and allows to elaborate the information obtained through searching the repository.

The integration of the SSQL paradigm into a distributed software artefact library [21] has generally shown, considering a rather limited experience , positive results, regarding both the efficiency of the searches and the impact of the methodology on the user.

These initial positive results have led to taking into consideration the possibility to improve some aspects of the SSQL system, which can be summarised as follows:

- the identification of the external repository is defined by the instruction createExternalLink : an interesting expansion of this mechanism is the possibility to identify and access different external repository in the same SSQL program

- the results returned from a query that accesses the repository, i.e. the identifiers of the retrieved elements, are saved in the constants' table inside the SSQL environment: it would be interesting to re-use the identifier returned by a query in order to state a new query, so that to improve the efficiency and the power of the search mechanism
- the possibility of tracking and recording the work done could be a useful functionality of the SSQL environment.

REFERENCES

1. AIT-KACI, H.,**The Wam: A (Real) Tutorial,** Paris Research Laboratory, January 1990.
2. JAYARAMAN, B., **Implementation of Subset Equational Programs**, JOURNAL OF LOGIC PROGRAMMING , 13(3), April 1992, pp. 299-324.
3. JAYARAMAN, B. and PLAISTED, D. A., **Functional Programming with Sets,** Third Int'l Conference on Functional Programming Languages and Computers Architecture, Portland, Oregon, 1987, pp. 194-210.
4. JAYARAMAN, B. and PLAISTED, D.A., **Semantics of Subset Logic Programming,** Technical Report, University of North Carolina at Chapel Hill, July 1988.
5. KOWALSKI, R.A., **Algorithm = Logic + Control**, COMMUNICATIONS OF THE ACM, 22(7), 1979, pp.424-436.
6. KOWALSKI, R.A.,**The Early Years of Logic Programming,** COMMUNIC-ATIONS OF THE ACM, 31(1), January 1988, pp.38-43.
7. NAIR, A., **Compilation of Subset-Logic Programs,** University of North Carolina at Chapel Hill, Master Thesis, December 1988.
8. PEYTON JONES, S.L.,**The Implement-ation of Functional Programming Languages,** PRENTICE- HALL INTERNATIONAL, Hemel Hempstead, UK, 1987.
9. STERLING, L. and SHAPIRO, E.Y., **The Art of Prolog**, MIT PRESS, Cambridge, March 1986.
10. SUCCI, G., **Set Representation in A Subset Equational Language**, State University of New York at Buffalo, Master Thesis, February 1991.
11. SUCCI, **G., La compilazione di linguaggi dichiarativi basati su insiemi per architetture parallele**, DIST - Universita\` di Genova, Ph.D Thesis, February 1993.
12. SUCCI, G. and MARINO, J., **A New Abstract Machine for Subset Equational Languages**, Technical Report, DIST - Universita\` di Genova, 1991.
13. WARREN, D.H.D., **An Abstract Prolog Instruction Set**, Technical Report, SRI International, Technical Note 309, 1983.
14. DURANTE, A. and BAUDINO, A., **Realizzazione di un compilatore Prolog per SEL**, Technical Report, Universita\` degli Studi di Genova, 1993.
15. BACKUS, J., **Can Programming Be Liberated from Von Neumann Style? A Functional Style and Its Algebra of Programs**, COMMUNICATIONS OF THE ACM, Vol. 21, No. 8, , August 1978, pp. 613-641.
16. COVINGTON, M.A., NUTE, D. and VELLINO, **Prolog Programming in Depth,** Scott, Foresman and C., 1988.
17. DOVIER, A., OMODEO, E.G., PONTELLI, E. and ROSSI, G., **A Logic Programming Language with Finite Sets,** Proceedings of the 8th Int'l Conference of Logic Programming, Paris, France, 1991.
18. VALERIO, A., **Interrogazione basata su asserzioni logiche d'insieme del sistema informativo distribuito Sarto**, Universita\` di Padova, Tesi di Laurea, 1995.
19. RUMBAAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F. and LORENSEN, W., **Object Oriented Modelling and Design**, PRENTICE-HALL, 1991.
20. SUCCI, G., MARINO, J. and COLLA, G.,**The SAM Instruction Set**, Proceedings of the Int'l Logic Programming Symposium 1991 Postconference 'Workshop, San Diego, California, November 1991.

21. SUCCI, G., VALERIO, A., CARDINO, G. BARUCHELLI, F. et al, **Sarto: A Configuration Management Tool for Distributed Multiorganizational Environments**, TIFF 1995, Trento, Italy.

22. FREUDENBERGER, S., SCHWARTZ J. and SHARIR, M., **Experience with the SETL Optimizer**, ACM - TRANS. ON PROGRAMMING LANGUAGES AND SYSTEMS, 5(1), January 1983, pp. 26-45.