# Title: "Glueing CASE Tools Together in a Heterogeneous CASE Environment"

**Authors:**
**Petra Luchner**
**Günter R. Koch**
**Franz Engelmann**

**Affiliation:**
**2i Industrial Informatics GmbH**
**Haierweg 20e**
**D-7800 Freiburg i.Br.**
**Telefon 07 61 / 4 22 57**
**Teletex 76 14 57 IND INF**
**Telefax 07 61 / 47 43 12**

Freiburg, den 20. April 1989 GK/DK/mo
File: CASE89

## Abstract

The integration of different existing tools, each of them not covering the whole life cycle, to a SEE (Software Engineering Environment) is an actual task.

There are two main mechanisms of integration.

The first one is the **vertical integration,** that means, that a common user interface and a common data base are available for the tools to be assembled into a CASE Environment.

The second one is the **horizontal integration** concerning the interface between tools for exchanging information. Here the semantics of the information is the mainly important part.

Either the interface can be an exclusive mutual link between two tools, which is an optimized but specific solution, or different tools can be connected by a Common Data Model for Integration, which is a general but very complex solution.

# Transforming LARS Requirements Specifications into High Level Target Languages

**Structure of the report**

?

# 1. Discussion of the leading idea

## 1.1 The rationales for developing a specific SARS- transformator system

Collecting and incrementally improving the very first descriptions of a future software system is a process which in practice suffers from the following deficiencies:

- The contractor providing the requirements has only poor and uncomplete ideas on the system he wants to be developed

- The provided specifications are never complete and stable. Many changements and additions permanently are made during the whole system design cycle.

- The contractor and the future users tend to refer as much as they are able to already existing and known solutions, i.e. they prefer to reuse former solutions. ( This is also a goal any system designer should try to achieve as far as possible).

- Most of the contractors as well as the designers of systems have some divergent knowledge on methods and tools for specifying and designing software, i.e. they have some understanding on how to model and how to design a system and they tend to impose their own constraints on methods, models and tools.

By consequence, the models and languages as well as the associated tools for describing (i.e. modelling) requirements show some relevant differences to classical and purely formal descriptions of the required functionality of a software system under design. The resulting description after a requirements specification may be

- incomplete by intention or by lack of information to be acquired in time.

- unstructured in a sense that the collected information does not yet show, how the future system will look like

- incoherent as many different types of suppliers of requirements information are involved.

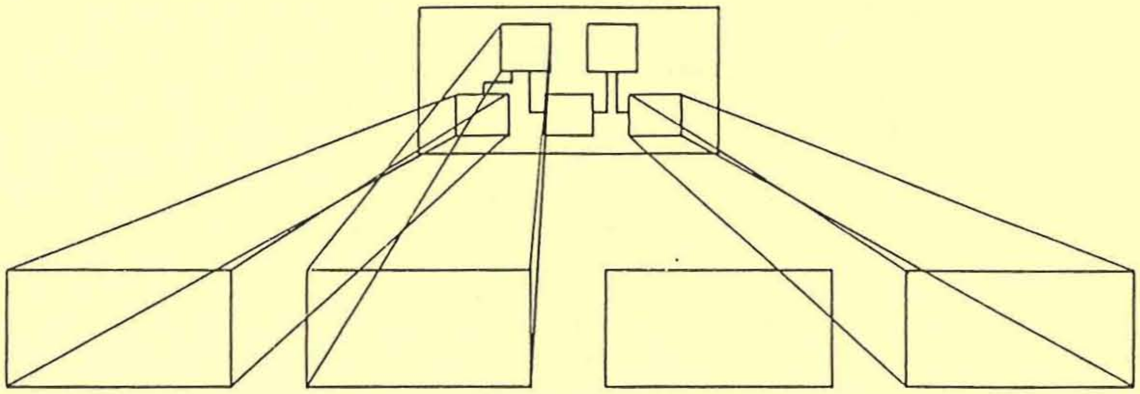
The language LARS and the associated tools in the SARS system [SARS] have been designed for meeting the needs on the one hand to collect and complete requirements definitions "smoothly" and incrementally and on the other hand to transform them into formal descriptions as extensive as possible. Thus LARS [LARS] and its associated method for writing LARS-specs will lead to some (hopefully) complete and consistent functional description of a system. However, this description by no means has an optimal structure in terms of an elegantly designed software system. The goal of requirements definition is not to design a system but to collect as many information on it as is available at the very early stage in the development life cycle.

Many requirement specification systems as e.g. EPOS [EPOS] claim, that the earliest description by means of homomorphic transformations can be mapped into a system design. Those concept-by-concept mappings propagate all the early fixed non-optimal structures into a design thus causing trouble in later phases, where e.g. performance deficiencies are detected to be originated by earlier design errors.

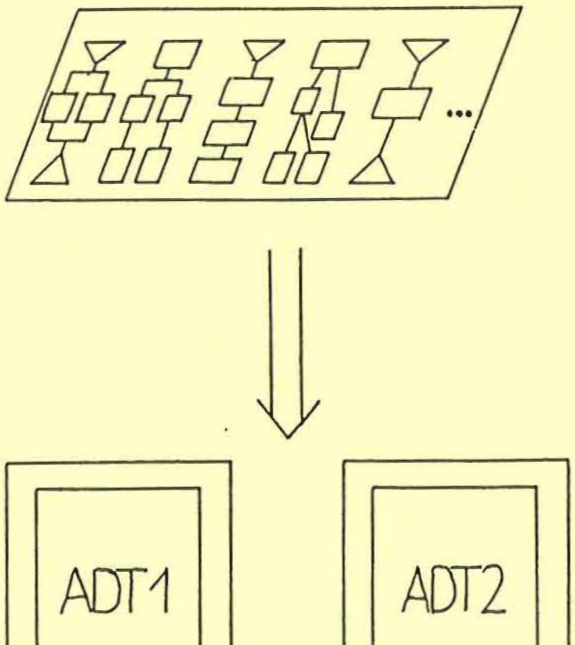
The position of the authors of this paper is, that requirements definition and software design are two activities, which are completely distinct and which have to be performed seperately. This means, that of course a requirements specification has to be transformed into a design description, however, this may not be a **homomorphic mapping**. Moreover, during the design an optimal model of the future system has to be elaborated, i.e. the design derived from the original requirements specs can be considered to be an optimisation process towards a system description done by means of a target language as e.g. a design language or a Very High Level Language like ADA.

The fundamental idea behind the SARTRE transformation tool described in section 2 is illustrated by fig. 1:

Fig.1: Traditional top-down decomposition approach:



Optimizing transformation by forming ADTs on the target side:



Whereas on the level of requirements specifications in LARS "classical" functions will be specified, a modern software design will be constructed by means of Abstract Date Types (ADTs) [Fell83]. In order to provide a design in the ADT style, LARS specs have to be analyzed such that data are collected e.g. into sets or classes and their associated operations have to be added to them, in order to provide complete ADTs. I.e. the design leads to a structure, which is completely different from the original description, which is considered to be a heteromorphic optimisation process. By consequence the mapping from the requirements description into the design description can not only be performed by formal transformations but also heuristical information will be needed to drive this optimisation process. A rule based approach therefore seems to be most appropriate.

## 1.2 References to earlier academic work

The discussion conducted under section 1 is not new in the academic world. Especially the mapping of stimulus-response modelled functions into an imperative language like ADA has been one subject of the PhD thesis of G. Persch on Transformational Development of Software [Per86]. Persch denotes any description of software under design as a document and he develops a very similar view on the design process as we discuss it here:

The different documents are formulated in languages which are designed for the phases, like stimulus-response-nets for requirements specification, algebraic specifications for system specification and Ada for implementation. The process of software development has to start from an idea of a software system and to generate the documents for the different phases [Leh83]. In the ideal case the documents of subsequent phases are transformations (in a very general sense) of the first documents in order to get more detailled, deterministic and executable descriptions of the software system [Bau81]. This transformational approach of software development has only been used informally in the past. New developments [Bro83], [Bal82] try to apply this approach with explicit support and sometimes in a rigorous way. Therefore we have to provide three prerequisits:

- Formal definitions of languages to express the documents of the transformations
- Definition of a transformation calculus which allows the formulation of transformation rules and derivations
- Development of support systems to automize transformations

The CIP-System [Par83] is a transformational system based on an algebraic wide-spectrum language. From a theoretical point the CIP-System has been a first step to provide the basis for the transformational approach. The implemented system however is rather restricted. It requires deterministic transformation rules and does not support automatic rule selection. Hence the programmer has to guide the transformations on a very low level.

The GIST-System [Bal82], [Fic85] is a transformational development system for a special specification language GIST. GIST is a very high level operational language with non-deterministic constructs (demons). Target language of the transformations is a special implementation language. The system is tailored to these languages. It provides a high degree of automation in respect to the rule selection but it is not based on a formal calculus.

An essential aspect of improved software development is the introduction of formal desription in the first phases (requirement specifications). This has been delayed because such documents could not be further processed. A formal requirements specification is only useful if it can be validated and if it is the basis for further developments. With transformation systems it is possible to transform formal requirements specifications into system designs and finally implementations. They also provide means to formulate specifications languages which are tailored to the application field. Such languages represent implicit knowledge about the applications which is then expressed explicitly with the help of the transformation system in the system design.

## 2. An Expert System Approach For Transformations

A transformation system to be used in a rule based transformational development environment is favourably constructed like an expert system. It consists of three subsystems, the rules, the inference machine, and the explanator.

### 2.1 The Rules

In general transformation rules have the form (cf.[Bau81]):

Input Pattern - Condition -> Output Pattern

with the semantics, that a structure which matches the input pattern is transformed to the output pattern if the conditions hold. The pattern and the condition share some variables which are instantiated during the match of the input structure. This general scheme is extended for our system by the following:

- Rule names are assigned to groups of rules. This introduces some kind of modularisation for rules and allows to reference rules.

- Rule names can have parameters to formulate meta rules which control the application of rules. By this mechanism rules can also be combined to larger rule sets.

- The parameters in rule names also allow to transport information from a global context to the place of application where it can be used in conditions and other applications.

- The patterns allow variables for function symbols and for parameter lists of functions.

- Output patterns can contain rule applications with the semantics of substituting them by one of the results of the invoked transformation. By such means a transformation rule can call subrules which are then composed to reach the required transformation.

The documents to be processed are represented as term data structure which corresponds directly to a abstract syntax representations of programs in compilers. This abstract syntax representation has to be provided by the database. The patterns in the transformation rules are formed in correspondence to it.

In our Prolog implementation of the transformation system, rules have the form:

**rule** rule_name: input_pattern => output_pattern **cond**
condition

Within the output_pattern and the conditions rule invocations are written as rule_name : input, patterns for functionals are written as variable#arguments. Rule names are terms with variables. Variables are marked by their beginning big letter. Conditions are conjunctions, disjunctions and negations of predicates.

**Example:** Rules for Expression Simplification

```
rule simple_exp: Exp => inner(simple): Exp.

rule simple : plus(X,0) => X.
rule simple : times(X,1) => X.
rule simple : OP#[C1,C2] => C3
cond constant(C1) and constant(C2) and eval(OP,C1,C2)=C3.

rule inner(R):F#[X] => R:F#[inner(R):X].
rule inner(R):F#[X,Y] => R:F#[inner(R):X,inner(R):Y].
rule inner(R):X => X cond atom(X).
```

The example consists of one top rule simple_exp which has two subrule bases simple and inner. The simple rules describe the different simplifications, whereas inner describes the strategy to apply these simplifications. Hence, inner can be seen as meta rule.

The third simple rule describes that any dyadic expression with constant arguments can be substituted by its result. Therefore we use a pattern for any dyadic expression with the operator OP. The inner rules have as parameter the name R of a rule base which is to be applied first to the innermost subexpression and then to enclosing expressions. Additionally we see how the output of inner is composed of the results of the invocations of inner to the subexpressions and the invocation of R to this composition.

## 2.2 The Inference Machine

The inference machine of an expert system has the task to apply the set of rules to the data. Hence it has to solve the following problems:

Pattern Matching-
Pattern matching is done by an extension of unification for first order logic (e.g. an extension of pattern matching in Prolog). The extension handles variables for function symbols and for lists of parameters.

Condition Evaluation-
If the input pattern matches the data the conditions have to be evaluated with the bindings done for the variables. Evaluation of conditions is handled like transformations to the boolean values. If they succeed the logical connectives are evaluated. Some of the conditions may require more sophisticated evaluation for example by a theorem prover. Additionally some of the conditions should be handled interactively, i.e. the programmer is asked for a decision.

Order of evaluation-

Most of the order of evaluation is already formulated by meta rules, which describe the rules to be applied to some data. However, there can be several rules (with the same name ) which can be applied. In this case a trial-and-error evaluation proceeds, i.e. the rules are applied in some order until one succeds.

Blind alleys-

If no more rule is applicable for a rule invocation, this can result from a wrong rule applied earlier. In this case the inference machine backtracks to a position where another rule can be applied. A transformation only succeeds if all rule invocations (which can invoke further rules ) are successful. Hence the transformation is goal-oriented. It backtracks from blind alleys automatically.

Termination-

A transformation can invoke rules for ever especially if there are commutativity and associativity rules. With the help of meta rules such rules be reformulated in a safe way, e.g. by applying commutativity only once to a term.

**Example:** Evaluation of expression simplification

```
simple_exp:times(a,1)
=> inner(simple):times(a,1)
=> simple:times#[inner(simple):a,inner(simple):1]
=> simple:times(a,1)
=> a
```

In the current implementation of the transformation system, the transformation rules are mapped to Prolog clauses and a standard Prolog interpreter builds the inference machine. Because of the efficient implementation of Prolog interpreters this implementation is useful. In the future a rule compiler (which is a set of transformation rules for transformation rules ) will be added which translates complete rule bases to programs which are called at the rule invocation.

## 2.3 The Explanator

The Explanator has the task to inform the programmer of the applied transformations. This is seperated into two cases:

Success -

If a transformation succeeds all the rules which have been applied are recorded in a history list. This list can be extended by the alternatives which are still possible. The list can be prepared for the user by suppression of the non interesting rules.

Failure -

If a rule invocation fails or the system can not prove a condition (or its contrary), the status has to be displayed in a way that the programmer can interact.

In the current Prolog implementation of the transformation system all applied rules are recorded and can be given during the transformation as a trace or afterwards as a history list. If during the evaluation of conditions undefined predicates are found, the system gives status information to the user like rule names and current part of the document and requires interactive use of the programmer.

## 3. An Example: A LARS-to-ADA Transformation

### 3.1 The Development Process

Stimulus-response-nets (SR-nets) as used in the SARS system are an established method for requirements definitions for process control systems [Alf80]. We use the language LARS [LARS] for the requirements specification. LARS is a modified variant of RSL which is used in SREM. For LARS exist a graphical editor and an analyzer, which allow a user-friendly preparation of requirements specifications. LARS specifications consist of three parts: a guard, which controls all the activities of the system, the SR-nets, which describe what action (response) is to be performed for a given event (stimulus), and the interfaces, which describe the interfaces to the technical process to be controlled. Data and its operations are described in natural language in LARS. Non-functional requirements like time constraint are already fixed in the LARS document.

The formalization of the data and its operations has to be done in the system specification phase by algebraic specifications [Gog84]. Data is described by abstract data types which consist of a signature and axioms in the form of conditional equations. The abstract data types include exceptions which can be raised in error situations and which can be handled by other operations. The first complete formal system specification is a mixture of LARS and algebraic specification. This document is the strating point for the transformation during the prototyping phase and the implementation phase.

During the prototyping phase the specification is transformed into an ADA program. Prototyping is applied very early to validate the system specification. The use of ADA as prototyping language and implementation has the advantage of having only one language in both phases and allows to reuse existing software within prototypes. The goal of the prototyping phase is an ADA program, which has the functional behaviour of the final system and which already handles the interfaces. It can be processed by an ADA compiler and ADA related tools [Per85].

The implementation phase concerns mostly with efficiency, user interfaces and embedding in the technical process. The implementation can have a different system structure than the prototype. It has to observe especially all non-functional requirements.

### 3.2 The similarity of concepts in LARS and ADA

LARS originally had been developed in the late seventies. The rationales of the language as to its richness in concepts are:

- It shall conform to the needs of engineers with scarce to no knowledge in computer science

- It shall be appropriate not only for functional specs but also for describing attributes and especially time conditions time constraints.

- The application areas for which LARS is suited are those of embedded real time systems, i.e. where computer and their software are one component of an overall system. Typical applications are:

  - control-systems within C/M
  - telecommunication functions
  - microelectronic components

- Many aspects and many specification paradigms shall be covered by LARS. Thus LARS is constituted by three sublanguages for describing INTERFACEs, Interconnections (GUARDs) and Functions (NETs and DATA desriptions).

- As LARS has to cover a variety of aspects and concepts, its richness is comparable to ADA. The following table gives an idea on the similarities and thus the potential mappings between LARS and ADA. However, this is only for demonstrating homomorphismic transformation. As mentioned in section one, the design of software systems seen as a process of optimizing transformations requires more sophisticated mappings.

| | | |
|---|---|---|
| GUARD | - | Task |
| NET | - | Task with START-Entry |
| INTERFACE | - | Task |
| Activation-Part | - | Select-Statement |
| START Net | - | Call Entry of the Net-Task |
| Terminate EVENT | - | Entry of the Guard-Task |
| Net-STRUCTURE | - | Taskbody starting with the ACCEPT of the Start-Entry |
| TERMINATE Net | - | Call Entry of the Guard-Task |
| ALPHA | - | Procedures, Functions |
| Interface EVENT | - | Interrupt-Entry |
| LARS-Data structures | - | ADA-Data structures |

Table 1: Mapping between LARS and ADA

## 3.3 The Transformation

The author now describes the general proceeding in the transformation and also shows some typical examples of transformation steps.

The generated prototype has to reflect the four parts of the LARS specification. The guard as central control of the system is transformed to an ADA task which waits for events from the interfaces, activates the SR-nets, and outputs data to the interfaces. The interfaces become entries of the guard task. The SR-nets which are parallel activities of the guard are transformed into tasks. Such a SR-net conceptually can be active several times in parallel. It can have local data which should be kept between several calls. Hence we have three possibilities for the transformation of SR-nets:

| | |
|---|---|
| Case(1) | a task object with permanent local data. |
| Case(2) | a task type where each SR-net activiation generates a new task object with temporary local data. |
| Case(3) | a task type within a package where each SR-net activiation generates a new task object with the global data in the package. |

The decision for one of the alternatives requires the knowledge of the existence of permanent data and parallel activations. The first fact can be derived by data flow analysis from the specification the second one is requested from the programmer.

**Example:** Rules for the Transformation of SR-Nets

```
rule net(GUARD):net(ID,GLOBALS,LOCALS,ALPHAS) %case(1)
            =>     [task(ID,entry('start'))
               ,task_body(ID,
                      decls:LOCALS,
                      net_body(GUARD):ALPHAS)]
            cond sequential_activation(GUARD,ID).

rule net(GUARD):net(ID,GLOBALS,LOCALS,ALPHAS) %case(2)
            =>     [task_type(ID,entry('start'))
               ,task_body(ID,
                      decls:LOCALS,
                      net_body(GUARD):ALPHAS)]
            cond not permanent_data(ALPHAS).
```

```
rule net(GUARD):net(ID,GLOBALS,LOCALS,ALPHAS) %case(3)
                =>[package(ID,
                       [task_type(ID,entry('start'))
                       |decls:LOCALS])
                ,package_body(ID,
                          task_body(ID,...))]
                cond permanent_data(ALPHAS).
```

15

The abstract data types are mapped into ADA (generic) packages. The signature can be transformed one-to-one into a package declaration. The axioms are mapped to function bodies which operate on the term algebra for the abstract data type. If we separate the operations into constructors and defined operations, the constructors build the term data structure and the defined operations manipulate this data structure as defined by the axioms[Per84]. By this technique the prototype implementation of the data types is automized.

In the implementation phase further transformations are applied to improve the abstract data type implementations. First rule bases for special data type implementations are evaluated. For example if there exist rule bases to implement sets by linked lists, by hash tables or by bit strings under certain conditions the system tries to evaluate which rule base is applicable for this specification. Further improvements can be done by evaluating whether there exists only one object of a type at a time. In this case the data is made local to the package and the functions are simplified by deleting the type from the parameter list.

Further transformations try to integrate parts of the system into one module with the help of global analysis. We describe two techniques, the combine and the pipe technique.

Combine [Dar81] expands several functions at their calling place and tries to combine them to a new function which avoids temporary data structures. A typical example is a function f1, which produces a sequence, and a function f2 which consumes this sequence. The combined function produces only one element and consumes it directly.

The pipe technique is an extension of the UNIX pipe mechanism. If some task t1 produces a sequence of elements which is consumed in the same order by some other task t2, then with the help of a buffer task tb, the construction of the sequence can be avoided. If t1 has produced an element it is given to tb, and if t2 will accept the next element tb gives it to t2. The evaluation of the applicability of these transformations is done on the specification level because the control and data flow analysis is much easier to handle there.

## 4. An Experiment Within ToolUse: Transforming LARS Into The Target Language OBER

### 4.1 OBER and ROMI

When different tools are to be combined, the problem on the semantics of the objects to be treated by the tools arises, as the tools have to understand on what objects they operate.

In order to generalize the description of objects to be manipulated such that different tools will have the same interpretation of the objects under work, an object model has been developed within one of the so called heterogeneous tool environments developed in W.Germany under the project name RASOP. The model is called RASOP OBJECT Model (ROM). It is described in more detail in [ROM]. Fig. 2 shows one part of the overall ROM-model.

For the formal description of the model itself the languages OBER (Object-Based Language for Engineering Requirements) has been defined within the RASOP project [OBER].

The OBER described objects are kept for administration in an object base, to which the tools can communicate by the means of the RASOP Object Manipulation Interface called ROMI [ROMI].

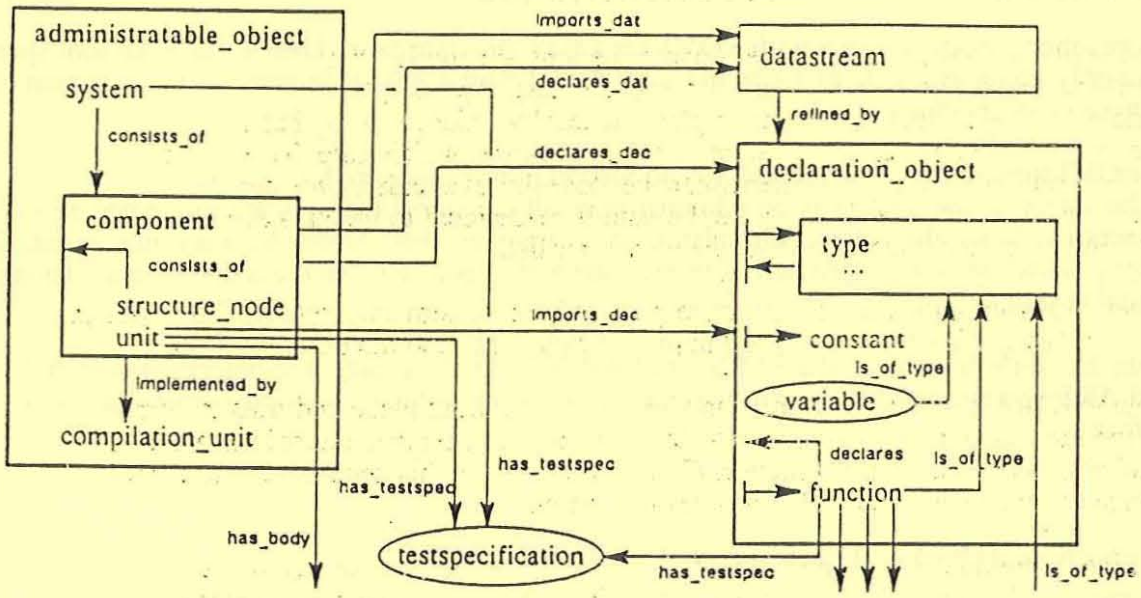


Fig.2: Part of the overall ROM-model

OBER is a language the definition of which was influenced by Chens Entity-Relationship-Model for modelling complex data structures as well as by the language SMALLTALK-80, from which the class-concept and the message concept was taken over. The OBER language has the following fundamental features:

- **Concept of aspects:**
  one tool only "sees" these classes and relations, which is necessary for its function. Thus an object may look different for different tools.

- **Description of objects:**
  Objects with the same characteristics are defined to be of the same class. The attributes of an object (i.e. as well of the class) define, from which atomar elements the objects were constructed. An attribute may also have semantic characteristics as e.g. a key attribute, a value, access rights etc.
  Building classes can be made hierarchically thus defining unheritance features.

- **Description of relationships:**
  Relationships may be defined between objects as well as between classes. Two basic types of relations are defined in OBER

  a) the structure-relationship defining the "is-part-of" relation between objects,

  b) the partner relationship defining relationships between independant objects.

Relationships may be defined as a multi tuple and the number of their instantiations (cardinality) may be restricted.

The ROMI (RASOP Object Manipulation Interface) is a programming interface implemented by procedures and declarations in C by which the creation, manipulation and deletion of objects may be performed by the tools within a RASOP environment.

The operations made possible with ROMI are based on concepts in OBER, with the tool specific aspect only being available to a specific tool. A tool may access an object via the selection of its attributes or its relations.

The ROMI functions are interpreted, i.e. no precompiling is required.
For the safety of the objects to be administrated, all changing accesses are implemented as long transactions. Thus the object administration guarantees that deadlocks may not occur, long working pauses lead to suspension of transaction and a tool may "tave over" a transaction started by another tool.

Within the RASOP-project, the SARS tool SARTRE has to manage a transformation from LARS into LARS-specific OBER-descriptions i.e. into objects which are transferable via the ROMI interface.

## 4.2 Some Sample Rules For Transforming LARS Into OBER

This chapter describes a short abstract of the SARTRE transformation rules which have been developed to build a part of the ROM-classes out of a LARS-specification.

In this early realisation phase of SARTRE it was an important goal to get some experiences with the underlying system. Therefore this rule-set has to be considered as a first transformational approach, infact not an optimized one, but can be accepted as a basis for stepwise improvements.

The first rule transforms the LARS-guard, the component, which describes the connections between the LARS-interfaces and the LARS-nets and which describes the data structures used by different LARS-nets.

For better understanding the original SARTRE-Syntax is not used in this abstract but a similar one, already used for the LARS-ADA transformation by G. Persch[Per86].

```
transfo_guard:

   guard ID; activation ACTIONS; common GLOBALS
   % Syntax of the LARS-guard

   -> unit ID

      has_body ACTIONS
       % The ACTIONS evaluate values of the LARS-interfaces
       % and activate the LARS-nets or send information to
       % the user.


      imports_dec functions(net_id: ACTIONS)
       % LARS-nets are transformed into functions.
       % To be able to call them out of this guard-
       % unit, their names must be imported.

      imports_dec functions input_interface_fct

      imports_dec_functions output_interface_fct
       % Evaluating the (LARS-) interfaces and sending
       % information to the user is done by calling
       % functions. These functions are declared in the
       % transformation goal of the LARS-interfaces. The
       % function names must be imported.
      end;


      unit global_data

         declares_dec variables GLOBALS

        % This unit declares the global data structures
        % which are used by different other units.
      end;
```

The second rule transforms a LARS-net into a function within a unit. This means that each LARS-net builds a unit with exactly one function. This way of modularisation is thought to be one of the parts which can be improved in further work.

```
transfo_net:

   net ID; structure NODES; common GLOBALS; private
   LOCALS;
   % Syntax of the LARS-net

   -> if: exchange_by_parameters: GLOBALS
       then:
       unit ID
         declares_dec function ID

         has_body (LOCALS, NODES)
           % The body of the function contains the
           % declarations of the local data and the
           % structure of the NODES.

         is_of_type (terminate, NODES)
           % The type of the function is the enumeration
           % of the% possible terminate stati of the function
           % (corresponding to the terminate nodes of the
           % LARS-net).

         has_sequence_of formal_parameters GLOBALS
           % Data exchanged with other functions are
           % transfered by formnal_parameters
   end;
```

Data, which are used in different functions and which the user do not want to be transfered by parameters, can be imported in the corresponding (net-)function.

## 4.3 First Experiences With SARTRE

The experiences with SARTRE we can refer are based on a transformation of a LARS-text into PASCAL-source code.
SARTRE is based on MProlog (the architecture is shown in Fig.3). MProlog and the TREX-rules were installed on a IBM-AT compatible Hardware. It turned out that, because of storage requirements we have to use a workstation-like Hardware for transforming larger texts. Furthermore the Prolog programming environment (PDSS) is not very comfortable to use (e.g. debugging aids, editor facilities etc.).

The experiences we gained until today are in this way that the given syntax of the TREX-rules is a useful feature for transforming one language into another.

We have no experiences how useful interactions between the user and the system can be integrated. That is due to further implementations.

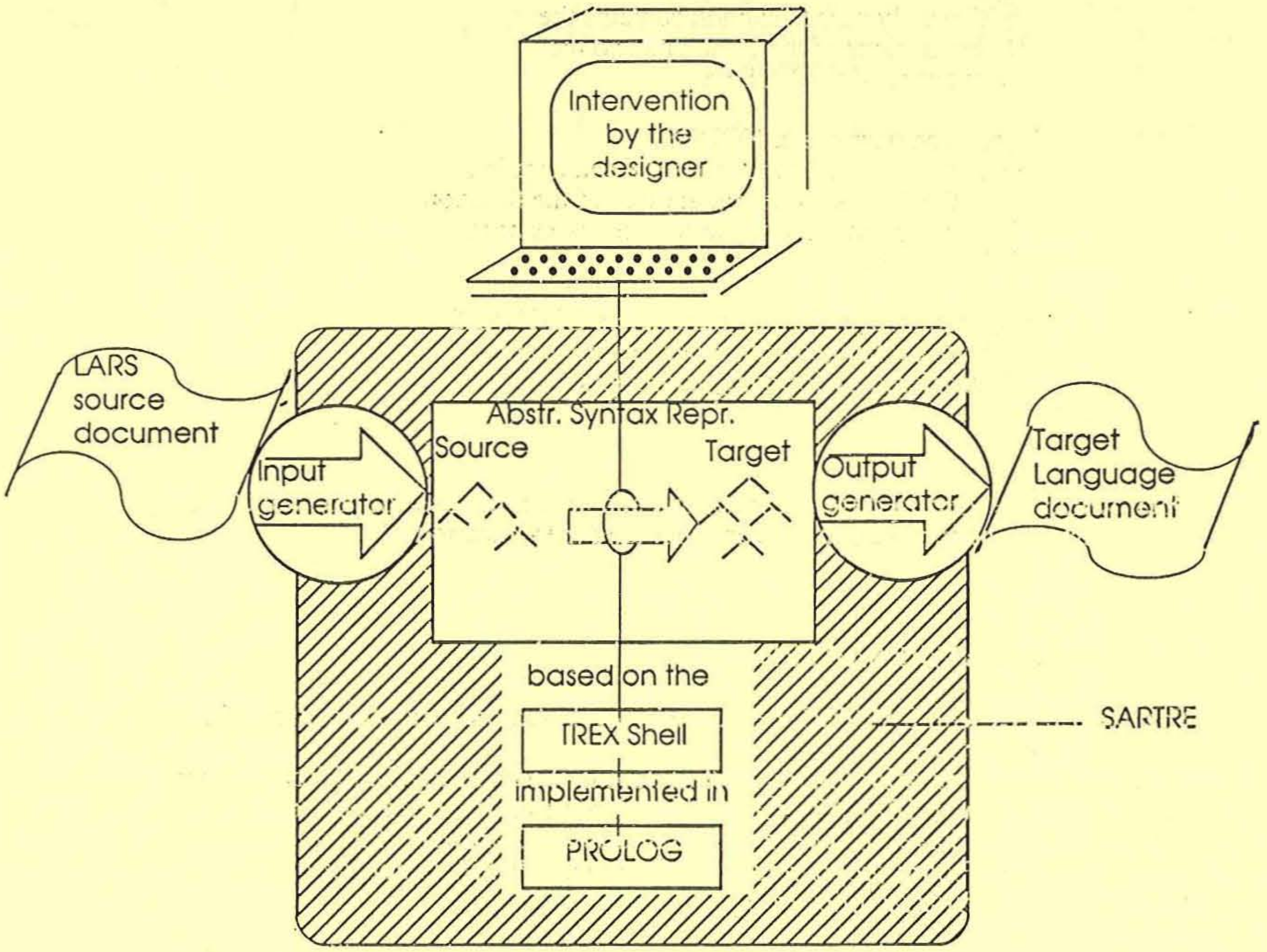


Fig.3: Architecture of SARTRE

## 5. The architecture of a SARS/SARTRE system

The authors intend to enrich their already existing SARS-system by tools for

- supporting the optimal LARS-to-Target Language transformation,
- giving guidance in collecting and completing requirements specifications,
- maintaining a library of reusable modules on target level.

The key role for such a system is played by SARTRE which will be parameterized for different target languages like PASCAL, C, MODULAII, ADA, PEARL and intermediate languages like OBER in its special use for describing ROM (see section 4.2).

An architectural chart of the goal of the SARS/SARTRE development is given by Fig.4 below.

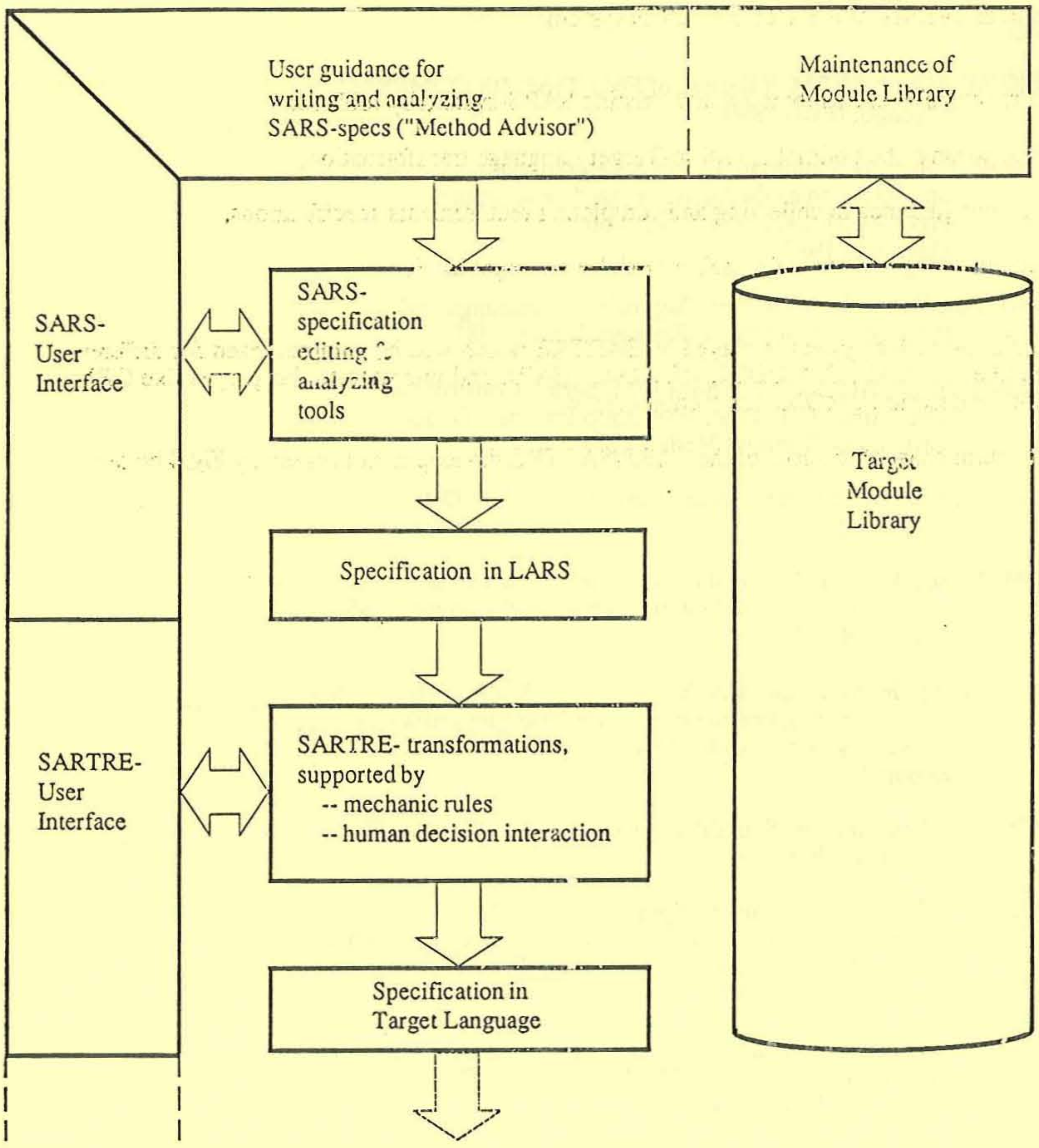


Fig.4: Architecture of the SARS/SARTRE system

**References**

Alf80 M. Alford: SREM at the age of Four, Proc. 4th COMPSAC, Chicago, IEEE 1980

Bal82 R. Balzer, N. Goldman, D. Wile: Operational Specification as the basis for rapid prototyping. Proc. ACM SIGSOFT Workshop on [illegible] Columbia 1982

Bau81 F.L. Bauer, K. Woessner: Algorithmic Language and Program Development, Springer Verlag 1981

Bro83 M. Broy: Algebraic Methods for Program Construction- The Project CIP, Proc. Workshop Program Transformations, Springer Verlag 1984

Chen76 P. Chen: The Entity/Relationship Approach to Data Modelling

Dar81 J. Darlington, R.M. Burstall: A System which Automatically Improves Programs, Proc. IFIP Congress 1981, North Holland

Epp83 W. Epple, M. Hagemann, M. Klump, G. Koch: SARS-System zur anwendungsorientierten Anforderungsspezi[illegible] University of Karlsruhe, Inst. f. Informatik III, Report 1983

EPOS Beschreibung des Spezifikationssystems EPOS, Fa. GPP, München, W. Germany

Fell83 M. Felleisen: Algebraische Spezifikationen als Ergänzung des Software-Entwicklungskonzeptes in SARS. Thesis of the University of Karlsruhe, Institute für Informatik III, 1983

Fic83 S.F. Fickas: Automating the Transformational Development of Software,USC-ISI/RR-83-108, Ph.D. Dissertation, Marina del Rey, 1983

Gog84 J.A. Goguen: Parameterized Programming, IEEE Software Engineering September 1984, 5/10

LARS LARS - Sprachbeschreibung, September 1987 2i Industrial Informatics, Freiburg, W.-Germany

Leh83 M.M. Lehmann, V. Stenning, W.M. Turski: Another look at Software Design Methodology,Imperial College London, Report DoC 83/13

OBER OBER-Object-Based Language for Engineering
Requirements, Version 1.0 from Oktober 1986
Not available in public

Par83 H. Partsch: The CIP Transformation System, Proc. Workshop Program Transformations, Springer Verlag F8-1984

Per84 G. Persch: Designing and S[illegible]ing Ada Software,
Proc. IFIP WG 2.[illegible], Canterbury, North Holland 1984

Per85 G. Persch et al: Karlsruhe Ada System - User Manual,
Gesellschaft für Mathe[illegible] und Datenverarbeitung mbH,
Report 1985

Per86 Ein Transformationssystem zur Programmentwicklung.
PhD Thesis at the University of Karlsruhe,
Faculty of Computer Science, 1986

ROM ROM - RASOP Object Model.
Version 1.0 from February 1987
Not available in public

ROMI ROMI [illegible] Manipulation Interface. Version 1.0
from June 1987. Not available in public

SARS [illegible] ·bung und Methodenbeschreibung.
[illegible] formatics, Freiburg
W.-G[illegible]