DESIGN AND EXPERIMENTATION OF AN EXPERT SYSTEM FOR PROGRAMMING IN-THE-LARGE

Giovanni Guida, Marco Guida, Sergio Gusmeroli, Marco Somalvico

Milan Polytechnic Artificial Intelligence Project
Politecnico di Milano
Milano, Italy

1. INTRODUCTION

The results of artificial intelligence research are often important to other areas than computer science itself. One area which presents a wide variety of potential applications of artificial intelligence tecniques, is the area of software production. A well known role of artificial intelligence in software technology has been in the area of program synthesis; several experimental systems based on different methodological approaches have been developed in the past (Barstow (1979); Bartels et al. (1981); Green (1977); Green and Barstow (1978); Green et al. (1979); Manna and Waldinger (1979); Smith (1981)). At the Milan Polytechnic Artificial Intelligence Project, the BIS system (Caio et al. (1982)) based on an approach oriented to problem reduction methodology for problem solving has been developed.
In our opinion, the recent evolution of knowledge-based systems is showing how the role of artificial intelligence can be further extended in dealing with the conceptual analysis of complex problems and applications. While the complexity of the problems solvable by a program synthesizer is of limited size, we may expect that a knowledge-based system can assist the designer of a complex software system devising its modular architecture. This activity is called within software technology programming in-the-large, as opposed to programming in-the-small, i.e. the classical programming activity devoted to design the data structures and algorithms needed for representing and solving a given problem.

The purpose of this paper is the illustration of the results obtained in a research project, devoted to design an expert system assisting the programmer in-the-large in his activity of problem analysis and software design: the ESAP (Expert System for Automatic Programming) (Guida et al. (1984); Guida et al. (1985)). The environment where the ESAP has been designed refers to a new rearrangement of the software life-cycle, in which several tools for automating software production are available. We call this environment Software Factory of the Future (SFF), as illustrated in Figure 1.

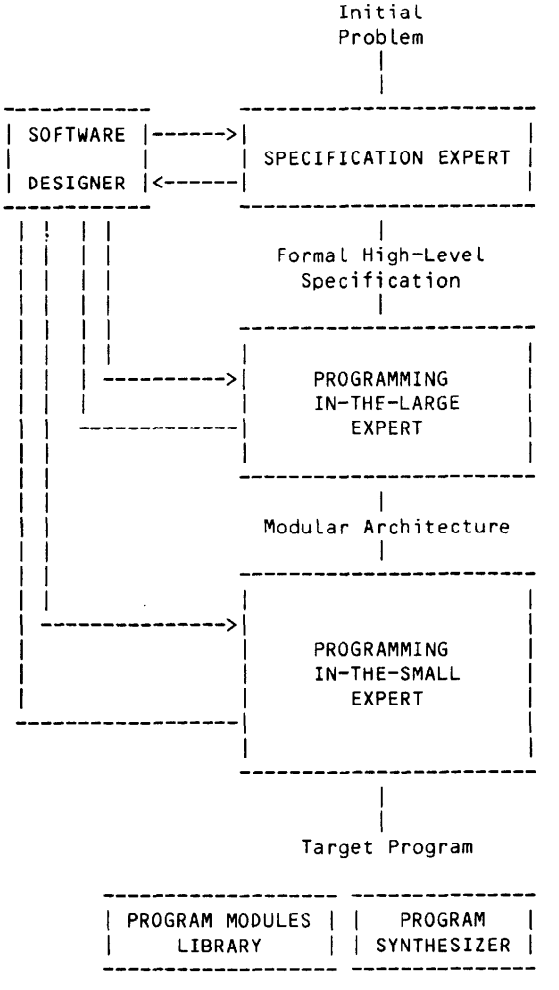


Fig. 1 The Software Factory of the Future.

ESAP receives in input the description of a large software system, supplied by the user by means of an high-level formal representation language and interacts with the programmer in refining the specification, progressively decomposing the problem into modules, and defining the appropriate interfaces among them.

This process halts when all the modules are simple enough to be easily manufactured by a set of lower-level actors for programming in-the-small, namely automatic program synthesizers and program modules libraries. The output of ESAP consists of the overall modular architecture of the desired software system, from which the final program can be implemented.

The goal of the project is, thus, to interactively support the activities carried out by a programmer during the design of a large software system. On the contrary, both the management of a software project and the programming in-the-small activity are not considered in this research.

ESAP has been implemented in Franz-Lisp, on DEC VAX11/780.

In this paper we will illustrate the experimental activity carried out in designing the various components of ESAP and experimenting with them. We will first discuss the architecture of ESAP (section 2); later, we describe the knowledge representation language (section 3) and the inferential engine (section 4); finally, the experimental results will be outlined (section 5).

2. ESAP ARCHITECTURE

In the past, several research directions have been carried out in the area of automatic programming (Barstow (1979); Green et al.(1979); Manna and Waldinger (1979); Smith (1981)). In particular, the approach of program transformation, which is based on incremental transformations of an high-level formal specification in order to achieve an automatically compilable representation (Manna and Waldinger (1979); Barstow (1984); Smith et al. (1985); Balzer (1985); Fickas (1985)), revealed very interesting. Furthermore, artificial intelligence techniques for the construction of knowledge-based systems have proved very powerful in constructing automatic programming systems (Barstow (1979); Green et al. (1979); Manna and Waldinger (1979)).

ESAP represents a synthesis of several approaches to program design and construction, developed in the areas both of automatic programming and of software engineering(Parnas (1972); Stevens et al. (1974); Parnas (1979); Booch (1983); Balzer (1985)), in the light of a new concept of the software life cycle, where several solutions in automating software production are available.
ESAP can support three major activities involved in software production:

- The **specification** of the problem to be solved and of the subproblems obtained during the decomposition process.
- The programming in-the-large, i.e. the **decomposition** of a problem into a set of cooperating subproblems, in order to obtain the correspondent modular architecture.
- The **programming** in-the-small, i.e. the solution of the simple problems constituting the leaves of the modular architecture.

ESAP is conceived as a set of expert systems, each one "expert" in one of the above mentioned domains. In particular, ESAP includes:

- **Specification Expert,** which enables the user to formally describe the initial problem he wants to solve, to modify the problem description and to incrementally complete it with new details. The activity of the specification expert divides into three phases:
  - Key-words (verbs and nouns) identification.
  - Description of a problem in terms of the classes of activities involved in it.
  - Problem reformulation by means of a formal representation language, in terms of input and output data and of operations on them and relationships among them.

  This approach to problem specification, supported by a knowledge base concerning the specification language (Representation language knowledge base), enables the user to gradually frame his problem in the ESAP environment, interactively supported during each step in the representation process. The output of the Specification Expert is a largely formal problem description, in which, however, the user is allowed to leave some informally expressed informations. At any stage during the decomposition process, the user can modify a representation, either substituting an informal sentence with a formal one, or adding previously omitted details.
  The last task of the Specification Expert consists in analyzing (syntactically and semantically) and translating the specification into an internal representation, easy to deal with by the inferential engine of ESAP.
- **Programming in-the-large Expert,** which analyzes a problem, finding out its possible decompositions in cooperating subproblems, shows them to the user, and, with his help, chooses the most promising one. Furthermore, a knowledge base concerning the specification language and the application domain allows ESAP to derive the specification of the subproblems, starting from the description of the initial problem and from the chosen decomposition.
  It relies on two different kinds of knowledge:

- software engineering knowledge, concerning principles and methodologies to be used in software design.
- domain knowledge, concerning criteria characteristic of the application domain to be used in leading the decomposition of problems into subproblems.

Both these knowledge-bases are independent of the specification language of ESAP and of the target programming language.

- **Lower-Level Actors Expert**, which analyzes a problem to find out whether and how it is solvable using the two foundamental resources ESAP has at his disposal:
  - A set of program synthesizers, varying in methodological approach and in application domain, capable of generating algorithms and constructing programs solving problems of limited complexity.
  - A set of program libraries, each one containing strongly parameterized programs, implementing elementary tasks in the chosen application domain. Each program has been designed and implemented with the aim to make it reusable for different, but similar tasks, using it directly or after performing on it some changes, according to user's needs. Program libraries are, thus, a collection of reusable software components. Each module is described by ESAP representation language and by the corresponding program, but the first description is the only one used by Lower-level Actors Expert.

The synthesis of the final program implementing the modular architecture and, thus, solving the user's problem is not at ESAP's charge, but it is performed by an external subsystem, called implementation expert.

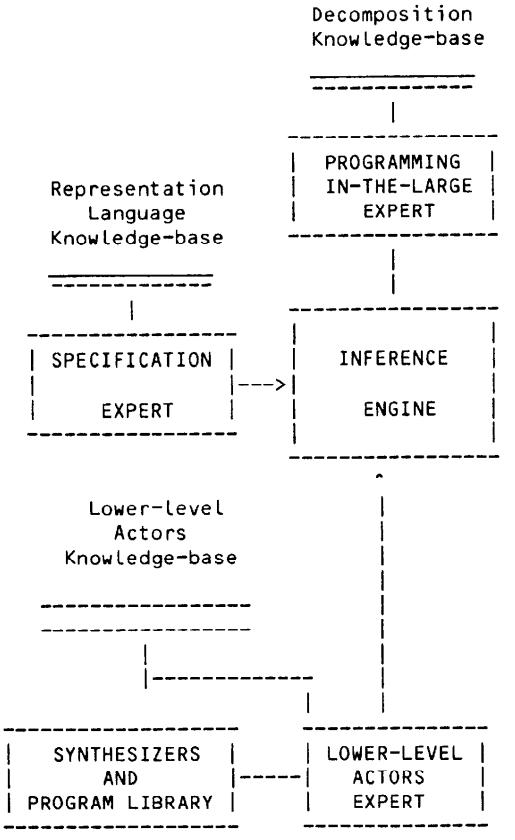


Fig. 2 The architecture of ESAP.

The architecture of ESAP (see Figure. 2) represents a prototype of an integrated environment for a first step towards toward the software factory of the future.

In particular, ESAP makes an extensive use of the crucial concept of reuse of resources, that we have interpreted in various ways:

- Reuse of general design methodologies, that is of domain and language independent principles (Stevens et al. (1974); Parnas (1972); Parnas (1979); Booch (1983)).
- Reuse of general domain concepts, both representation and target programming languages independent (Barstow (1984); Barstow (1985); Kant (1985); Adelson and Soloway (1985)).
- Reuse of software components, i.e. of parameterized and ad hoc designed programs, developed in order to solve different, though similar problems (Sommerville (1982); Wegner (1984); Ramamoorthy et al. (1984)).

3. THE KNOWLEDGE REPRESENTATION LANGUAGE

We have identified two representation languages in ESAP: one for the interaction with the user, allowing semplicity of use and supporting abstractions in the definition of the problem he wants to solve; the other for the inferential activity of the system, allowing an efficient representation of facts and rules.
The possibility to translate the representation of a problem from the former language to the latter is offered by a conversion algorithm.

3.1 User representation language

The foundamental characteristic of ESAP user representation language is the ability to support abstractions in the definition of a problem. This goal is achieved by allowing the user to neglect all the details of the representation that he considers unimportant and to define them only when it is necessary to complete the description.
Furthermore, the user is allowed to employ terminology and concepts of the task domain, since the system has knowledge about them and about how to deal with.

The interaction with the user through ESAP user representation language is supported at two abstraction levels:

- **Class definition level**, which allows the definition of a problem in terms of the types (classes) of the functionalities involved in it. This allows a graceful approach to the problem, that is represented in an highly abstracted way.

  As an example,

  ```
  class 1  COMPUTE interest IN current_account
  class 2  UPDATE archiv IN current_account
  ```

  represents the problem consisting in computing the interest of all current_accounts of a bank and in updating the corresponding archivs with the new balance computed values. The user can disregard all the details not relevant for a first description of his problem, as the value of the interest rate or the time period on which compute the interest.

- **Module definition level**, which allows an incremental representation of the problem through a set of primitive expressions. They make reference to an abstract data type, called "archiv", whose instances represent the foundamental data of all the problems in a bank environment.
  There are three basic instances of the "archiv" type: a current account archiv, containing informations about the present situation of the accounts; a transactions archiv, recording all the transactions on the accounts; a registry archiv, containing the private data of the clients.
  Some primitives are inspired by the relational algebra and they allow to handle archivs by means of operations of retrieving, selecting, projecting on fields, joining, and so on.
  Other primitives define the usual set operations of union, intersection and difference of archivs.
  Other more primitives allow to define updating operations, consisting in inserting, deleting or modifying fields or tuplas into an archiv.
  Furthermore, we have introduced the "compute" and "compact" primitives: the former allows to represent typical computations in the management of current accounts, using domain concepts. The latter allows two types of manipulation of an archiv: adding the values of an assigned field of the tuplas with the same value of another assigned field and adding the values of an assigned field in all the archiv.
  The primitive "ord" allows to sort the tuplas of archivs on the basis of values assumed by their fields or by expressions containing one or more references to fields.
  Finally, we have introduced the primitive "reduce", which allows to select occurrences of elements on the basis of ordinal constraints.

The example above may be detailed at the module level as follows:

```
MOD esmp (c_a_archiv trans_archiv
          rate date_1 date_2
          --> new_c_a_archiv)
LINK
BODY INPUT  c_a_archiv :
                current_account_archiv
            trans_archiv :
                transactions_archiv
            rate : number
            date_1 : date
            date_2 : date
     OUTPUT new_c_a_archiv :
                current_account_archiv
     STOP
     COMPUTE interest OF current_account
      FROM date_1 TO date_2
                WITH_ANNUAL_RATE rate %
      WHERE (USING c_a_archiv
                AND trans_archiv)
         IN c_a_archiv (money = dollar)
         IN trans_archiv (money = dollar)
     END
     MODIFY_FIELD (balance) INTO c_a_archiv
      WITH_NEW_VALUE ((c_a_archiv ^ balance
                + interest))
      WHERE (OBTAINING new_c_a_archiv)
     END
MOD_END
```

In this example, we have already introduced all the particulars of the problem, but ESAP is able to start its activity even with incomplete specifications, allowing the user to correct and modify the representation at each step of the decomposition process. As an example, we can omit the where conditions (i.e. the conditions appearing after the key-word "IN", which define the tuplas to be handled by the primitive), but the system is nevertheless able to suggest a set of possible decompositions.
The flexibility of this language is increased by means of informal user sentences, which can be included in the specification of a problem, allowing to satisfy particular user's exigencies. These sentences will be handled interactively by the system in a second step, in order to transform them into formal expressions containing known concepts and in acquiring new knowledge.

3.2 System representation language

A knowledge base may be viewed as made up of facts and actions (Hayes-Roth (1983); Laurent (1984)). In the ESAP system representation language, a fact is a pattern with the following structure:

FACT --> object (relation object | attribute value)$^{+}$

where "object" may be any element of the description of the activity of a module. An object may have one or more attribute with a value (e.g. we represent the fact that a variable A is of type T as: (A type T)). Furthermore, an object may be bound to another object by a relation (e.g., we represent the fact that the variable A is input for the module M as: (M input A)).
The left-hand side of a rule is made up of conjunctions, disjunctions and negations of patterns; the right-hand side is made up of new patterns to be added to the knowledge base, when the rule is applied. Furthermore, we allow to use in the left-hand side of a rule two special patterns, namely ASK and ASKY/N. They allow to point directly to functions that evaluate to a boolean value (ASKY/N) or to a new binding for the variables (ASK).

4. THE INFERENCE ENGINE

As we have pointed out in section 2, ESAP is conceived as a set of knowledge-based subsystems. The control cycle, on which the system is based, determins the correct order in which each expert has to be activated.

We have considered a particular interpretation of an expert system in terms of the state space model for problem solving (Laurent (1984)). In this view, a given knowledge base represents a state and an action represents an operator that allows a transition from a state to another state. In particular, the inner representation of the initial user's problem represents the initial state.
The control cycle of an expert system is usually based on four steps: selecting the next state to be expanded; finding all the transitions appliable to the chosen state; selecting the next transition to apply; effectively applying the chosen transition.
The conflict resolution may be implemented in two sub-steps, consisting in the choice of the object on which to apply the next transition and in the choice of the transition to be executed; the two steps can appear in any order in the control cycle of the inferential engine.
In particular, we have chosen a S-O-A (State-Object-Action) strategy, consisting in selecting first the next state to be expanded, then the object (i.e. the module) on which the expansion will be based, finally the action (i.e. the decomposition operator) to be applied on it.
A state is a set of modules, that are leaves of the decomposition tree under development. At each control cycle, the system checks for the necessity to change the state. This corresponds to abandoning the current decomposition and restoring a previous state, in which it is possible to apply a new decomposition operator to derive a different modularization of the same problem (state selection).
The next step consists in activating the programming-in-the small expert to check the terminality of the leaves of the modular architecture and to choose the next one to decompose (object selection).
Then, the programming in-the-large expert searches for all the elementary appliable decomposition operators, reasoning on them and selecting one of their consistent and complete combinations, possibly the most promising one (action selection).
Finally, the choosen operator is applied and the representation expert is activated to deduce the representation of the subproblems from the initial problem's one and to complete them, interacting with the user. This activity corresponds to the construction of the new current state for the next control cycle of the inferential engine of ESAP. Each cycle corresponds to a design step for the construction of the modular architecture of the desired software system.

The inferential process is based on production rules and on metarules, that define the order in which a set of goals are to be achieved (i.e. implement strategies) and allow to quickly focouse on relevant rules' subsets, referring to them by name (Aiello (1983)).

5. EXPERIMENTAL RESULTS

ESAP has been successfully implemented at the Milan Polytechnic Artificial Intelligence Project on DEC-VAX11/780, in Franz Lisp.
The task domain we have chosen is that one of

current accounts management in a bank. This domain is sufficiently known and large to allow a realistic programming in-the-large activity. At the present, in the Knowledge Base of ESAP there is enough knowledge to deal with a set of domain concepts as "interest", "interest rate", "current account", "transactions", and so on. Furthermore, domain-independent knowledge has been supplied to deal with a top-down structured design methodology for the development of the modular architecture of software systems. It is currently under development a knowledge area to support object-oriented design.
Such programming in-the-large knowledge allows ESAP to find out all the possible decomposition criteria that are applicable to a given problem. The final choice among the set of applicable operators is at user's charge, but ESAP can give suggestions to direct the decomposition towards problems manageable by its lower level actors for programming in-the-small.

Starting from the example outlined in section 3, we will sketch the decomposition process.

Let's consider the following top-down design rules:

```
RULE_1

IF   A module $M has a subactivity $S1
                and
     The module $M has a second subactivity $S2
                and
     A variable $V is output for $S1
                and
     The same variable $V is input for $S2

THEN The module A is sequentially bound by
     $S1 and $S2 through $V


RULE_2

IF   A module $M is sequentially bound by
     $S1 and $S2 through $V
                and

THEN The module $M produces a submodule
                derived from $S1
     The module $M produces a submodule
                derived from $S2
     The variable $V represents  the
     interconnection between the two
                submodules.
```

Each module representation is translated into an internal form, expressed as a set of facts with the structure described in section 3.2. These facts are matched against the patterns of the left-hand side of the rules during the activity of the system.
The application of RULE_1 and RULE_2 on the set of facts derived from the representation of the sample problem allows to add to the knowledge base the basic informations to be used by the representation expert to derive the description of the submodules. Acting on its private knowledge area and interacting with the user to ask for informations whenever they cannot be automatically deduced (e.g. asking for the names of the new modules or of newly introduced variables), the representation expert produces the following descriptions:

```
MOD comp_int (c_a_archiv trans_archiv
                rate date_1 date_2
              --> interest_list)
LINK EXPORT interest_list
BODY INPUT  c_a_archiv :
                current_account_archiv
            trans_archiv :
                transactions_archiv
            rate : number
            date_1 : date
            date_2 : date
     OUTPUT interest_list :
                (arch c_a_num : c_a_number
                 total_interest : number)
     STOP
     COMPUTE interest OF current_account
      FROM date_1 TO date_2
                WITH_ANNUAL_RATE rate %
      WHERE (USING c_a_archiv AND
                          trans_archiv)
         IN c_a_archiv (money = dollar)
         IN trans_archiv (money = dollar)
            (OBTAINING interest_list)
     END
MOD_END


MOD modify_archiv (c_a_archiv interest_list
                    --> new_c_a_archiv)
LINK IMPORT (interest_list AS interest_list
                          FROM comp_int)
BODY INPUT  c_a_archiv :
                current_account_archiv
            interest_list :
                (arch c_a_num : c_a_number
                 total_interest : number)
     OUTPUT new_c_a_archiv :
                current_account_archiv
     STOP
     MODIFY_FIELD (balance) INTO c_a_archiv
      WITH_NEW_VALUE ((c_a_archiv ^ balance +
            interest_list ^ total_interest))
      WHERE (JUNCTION c_a_archiv ^ c_a_num =
                      interest_list ^ c_a_num)
            (OBTAINING new_c_a_archiv)
     END
MOD_END
```

As shown above, ESAP interprets the approach of program transformation breaking a complex problem into co-operating subproblems, in such a way to allow to automatically define

the structure of the modular architecture to be constructed. This may be viewed as an automatic documentation of the developement process of the software system solving the initial problem.
At this stage, the Lower-level Actors Expert is able to automatically establish that the module "modify_archiv" matches against a module of the library, and thus it advises the user that the only module "comp_int" is to be decomposed. The module "comp_int" is now functionally bound, i.e. it defines a unique activity, with a well precised output. Nevertheless, the decomposition process can continue by applying on the module domain-oriented rules, to further reduce it to simpler subproblems.
The following two rules are now applicable on the module "comp_int":

```
RULE_3

IF   A module $M has only one subactivity $S1
                and
     The subactivity $S1 has where_conditions
     on archiv $A

THEN The module $M produces a
                submodule reducing
     the archiv $A on the basis of
                the where_conditions
     The module $M produces a
                submodule derived from $S1
     without where_conditions


RULE_4

IF   A module $M has only one subactivity $S1
                        and
     The subactivity $S1 is of kind compute
                        and
     The object of subactivity $S1 is
     interest
                        and
     The environment of subactivity $S1
     is current_account
                        and

THEN The module $M produces a submodule
     calculating the interest of the
     balances at closing date
     The module $M produces a submodule
     calculating the interest of the
     transactions
     The module $M produces a submodule to
     combine the results of the two
     submodules above
```

Choosing the RULE_3, we obtain the three following interconnected submodules:

```
MOD red_acc_archiv (c_a_archiv -->
                red_c_a_archiv)
LINK EXPORT red_c_a_archiv
BODY INPUT c_a_archiv :
                current_account_archiv
     OUTPUT red_c_a_archiv :
                current_account_archiv
     STOP
     SELECT (( c_a_archiv ^ money = dollar)
             OBT red_c_a_archiv)
     END
MOD_END



MOD red_tr_archiv (trans_archiv -->
                red_trans_archiv)
LINK EXPORT red_trans_archiv
BODY INPUT trans_archiv :
                transactions_archiv
     OUTPUT red_trans_archiv :
                transactions_archiv
     STOP
     SELECT (( trans_archiv ^ money = dollar)
             OBT red_trans_archiv)
     END
MOD_END



MOD comp_int_son (red_c_a_archiv
                red_trans_archiv rate
                  date_1 date_2  -->
                interest_list)
LINK IMPORT (red_c_a_archiv as
                red_c_a_archiv
                from red_acc_archiv)
            (red_trans_archiv as
                red_trans_archiv
                from red_tr_archiv)
    EXPORT interest_list
BODY INPUT  red_c_a_archiv :
                current_account_archiv
            red_trans_archiv :
                transactions_archiv
            rate : number
            date_1 : date
            date_2 : date
     OUTPUT interest_list :
                (arch c_a_num : c_a_number
                  total_interest : number)
     STOP
     COMPUTE interest OF current_account
      FROM date_1 TO date_2
                WITH_ANNUAL_RATE rate %
      WHERE (USING c_a_archiv AND trans_archiv)
            (OBTAINING interest_list)
     END
MOD_END
```

The first two modules define the where_conditions' reduction in the specification and they are solvable by the lower-level actors. The third one represents the father-module, without the where_conditions and it needs further steps of decomposition.

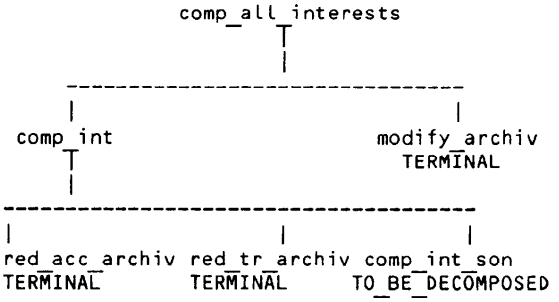
Choosing, instead, RULE_4, we obtain other three interconnected submodules:

```
MOD comp_int_of_balance (c_a_archiv
```

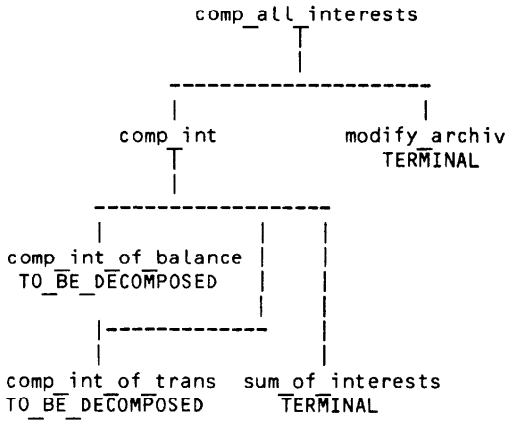
```
                        trans_archiv
                        rate date_1 date_2
                        --> int_of_balances)
LINK EXPORT int_of_balances
BODY INPUT  c_a_archiv :
                current_account_archiv
            trans_archiv :
                transactions_archiv
            rate : number
            date_1 : date
            date_2 : date
     OUTPUT int_of_balances :
                (arch c_a_num : c_a_number
                 balance_int : number)
     STOP
     COMPUTE interest OF balance
                FROM date_1 TO date_2
      WITH_ANNUAL_RATE rate %
      WHERE (USING c_a_archiv
                AND trans_archiv)
         IN c_a_archiv (money = dollar)
         IN trans_archiv (money = dollar)
            (OBTAINING int_of_balances)
     END
MOD_END


MOD comp_int_of_trans
        (trans_archiv rate date_1 date_2
         --> int_of_trans)
LINK EXPORT int_of_trans
BODY INPUT  trans_archiv :
                transactions_archiv
            rate : number
            date_1 : date
            date_2 : date
     OUTPUT int_of_trans :
                (arch c_a_num : c_a_number
                 trans_int : relativ)
     STOP
     COMPUTE interest OF transactions
        FROM date_1 TO date_2
        WITH_ANNUAL_RATE rate %
       WHERE (USING trans_archiv)
         IN trans_archiv (money = dollar)
            (OBTAINING int_of_trans)
     END
MOD_END


MOD sum_of_interests
        (int_of_balances int_of_trans
         --> interest_list)
LINK IMPORT (int_of_balances
                as int_of_balances
                from  comp_int_of_balance)
            (int_of_trans as int_of_trans
                from comp_int_of_trans)
     EXPORT interest_list
BODY INPUT  int_of_balances :
                (arch c_a_num : c_a_number
                 balance_int : number)
            int_of_trans :
                (arch c_a_num : c_a_number
                 trans_int : relativ)
     OUTPUT interest_list :
                (arch c_a_num : c_a_number
                 total_interest : number)
     STOP
     MODIFY_FIELD (balance_int)
                INTO int_of_balances
       WITH_NEW_VALUE (
          ( int_of_balances ^ balance_int +
            int_of_trans ^ trans_int ))
       WHERE (JUNCTION
                int_of_balances ^ c_a_number =
                int_of_trans ^ c_a_number)
             (OBTAINING interest_list)
     END
MOD_END
```

The above decomposition implements a technique, namely the "Direct Method", used in bank environment, in order to compute the interests of the current accounts. It consists first in computing the interests of the balances at the closing date, then in computing the interests of the transactions and finally in adding them. The first two modules can be further decomposed by the system; the third one, instead, is considered solvable by meaans of the lower-level actors.

During its activity, ESAP suggests to the user several alternative decompositions (see Figure. 3) of the same problem, leaving the choice of the most promising one at user's charge.



a. RULE_3 application.



b. RULE_4 application.

Fig. 3 First steps of alternative decompositions (a. and b.).

In the above example, the application of RULE_3 allows to achieve a simplier decomposition of the problem, taking advantage of the capabilities offered by the lower-level actors, i.e. by the program synthesizer BIS and by the program library.
The decomposition process halts when there are not operators to further reduce the subproblems.

Particular attention has been devoted to the integration in ESAP environment of both program libraries of reusable software components and program synthesizers. This has lead us to the conception and the development of the Lower-level Actors expert.
At the present, only the BIS (BIdirectional Synthesizer) system is at ESAP disposal. Given a simple problem, ESAP checks for its synthesizability by BIS, analyzing the description in the ESAP user representation language. The analysis is based on a knowledge area that takes into account the properties of ESAP representation language and of the BIS one, first of all checking for the possibility of translation from the former to the latter.
Determining quantitatively the complexity level of a problem and comparing it with the solving power of a program synthesizer is a very difficult and challenging task. On the other hand, it seems to us too shallow to consider a program synthesizer ideal, i.e. actually able to derive a program solving any problem representable in terms of its specification language.
For these reasons, we adopted an intermediate solution, consisting in qualitative reasoning about the synthesizability of a problem by BIS: on the basis of our experimentations with the BIS system, we found a set of conditions to be satisfied by a problem specification in order to make the corresponding algorithm efficiently implementable. Thus, the system is able to check for the satisfaction of such a set of conditions, in order to establish the possibility to solve by BIS a given problem.
We consider this aspect of ESAP of conceptual relevance, since it offers the opportunity to deal within a unique system with different types of knowledge representation languages. Intermediate knowledge areas give the possibility to reason on the various representation languages, providing an adeguate interface among them and supporting an eventual translation from one to another.

The other lower level actor for programming in-the-small at ESAP disposal is the program library, i.e. a collection of reusable software components. The representation of a given user problem of low complexity and a module in the library may be isomorphic, if they have the same description in the user representation language. They are alike if they have syntactically and semantic different representation, but if the program can be modified to meet the user needs.
If a problem is isomorphic to or like a given library module, it can be automatically manufactered by ESAP. If it is not the case and if no decomposition operators are applicable, the code writing is leaved at user's charge.
The match against the program library may occur after a trasformation of the user specification into a semantically equivalent one, more convenient for the matching process.
The management of the program library is based on a knowledge area, taking into account meaning preserving transformations within ESAP user representation language. Furthermore, this knowledge area contains rules that define elementary modifications that it is possible to implement on an already existing program. The programming in-the-small expert allows the user to reason on programs at the abstract level of the representation language rather than at the concrete level of a particular programming language.

6. CONCLUSION

The goal of the ESAP project, carried out in the last two years, has been the development of a prototype system for the software factory of the future.
The realization of our system has pointed out some interesting research areas related to ESAP project. In particular, it is our opinion that attention has to be devoted to the development of a subsystem for the acquisition of new domain knowledge and for its integration with the existing one. This will allow ESAP to learn new task-domain concepts, their meaning, their properties, and how to deal with problems including such concepts (i.e., how to decompose them).

REFERENCES

1. Adelson, B. and Soloway, E. (1985). The role of domain experience in software design. IEEE Trans. on Software Engineering SE-11 (11), 1351-1360.
2. Aiello, L. (1984). The uses of meta-knowledge in AI systems. In T.O'Shea (Ed.)ECAI '84 Advances in Artificial Intelligence. Elsevier, Amsterdam, NL.
3. Balzer, R. (1985). A 15 year perspective on Automatic Programming. IEEE Trans. on Software Engineering SE-11 (11), 1257-1267.
4. Barstow, D. (1979). Knowledge-based program construction. North-Holland, Amsterdam, NL.
5. Barstow, D. (1984). A perspective on Automatic Programming. The AI Magazine 5(1), 5-27.
6. Barstow, D. (1985). Domain-specific Automatic Programming. IEEE Trans. on Software Engineering SE-11 (11), 1321-1336.
7. Bartels, U., Olthoff, W. and Raulefs, P. (1981). APE: An expert system for Automatic Programming from abstract specifications of data types and algorithms. Proc. 7th IJCAI, Vancouver, BC, Canada, 1037-1043.
8. Booch, G. (1983). Software Engineering with ADA. Addison-Wesley publ., Amsterdam, NL.
9. Caio, F., Guida, G. and Somalvico, M. (1982). Problem Solving as a basis for program synthesis: design and experimentation of the BIS system. Int. Journal on Man-Machine Studies 17, 173-188.
10. Fickas, S.F. (1985). Automating the transformational development of Software. IEEE Trans on Software Engineering SE-11 (11), 1268-1277.
11. Green, C. (1977). A summary of the PSI program synthesis system. Proc. 5th IJCAI, Cambridge, MA, 380-381.
12. Green, C. and Barstow, D. (1978). On program synthesis knowledge. Artificial Intelligence 10(3), 241-279.
13. Green, C. et al. (1979). Results in knowledge based program synthesis.Proc. 6th IJCAI, Tokyo, Japan, 342-344.
14. Guida, G., Guida, M., Gusmeroli, S., and Somalvico, M. (1984). ESAP: an Expert System for Automatic Programming. In T. O'Shea (Ed.), ECAI '84: Advanced in Artificial Intelligence, Elsevier, Amsterdam, NL, 585-588.
15. Guida, M., Gusmeroli, S. and Somalvico, M. (1985). ESAP: an intelligent assistant for the design of software systems. Proc. Cognitiva '85, Paris, F, 201-209.
16. Hayes-Roth, F., Waterman, D.A., and Lenat, D.B. (Eds.)(1983). Building Expert Systems. Addison-Wesley, Reding, MA.
17. Kant, E. (1985). Understanding and automating algorithm design. Proc 9th IJCAI. Los Angeles, CA, 1243-1253.
18. Laurent, J.P. (1984). Control structures in expert systems. Technology and Science of Informatics 3(3), 147-162.
19. Manna, Z. and Waldinger, R. (1979). Synthesis: Dreams ==> Programs. IEEE Trans. on Software Engineering SE-5(4), 294-328.
20. Parnas, D.L. (1972). On the criteria to be used in decomposing systems into modules.Comm. ACM, 1053-1058.
21. Parnas, D.L. (1979). Designing software for ease of extensions and contraction. IEEE Trans. on Software Engineering SE-5(2), 128-138.
22. Ramamoorthy, C.V. et al. (1984). Software Engineering: problems and perspectives. IEEE Trans. on Software Engineering, 191-209.
23. Smith, D.R. (1981). A design for an automatic programming system. Proc. 7th IJCAI, Vancouver, BC, Canada, 1027-1029.
24. Smith, D.R., Kotik, G.B., and Westfold, S.J. (1985). Research on Knowledge-Based Software Environments at Kestrel Institute. IEEE Trans. on Software Engineering SE-11(11), 1278-1295.
25. Sommerville, I. (1982).Software Engineering. Addison-Wesley, Reding, MA.
26. Stevens, W.P., Myers, G.J., and Constantine, L.L. (1974). Structured design. IBM System Journal 13(2), 115-137.
27. Wegner, P. (1984). Capital intensive software technology. IEEE Trans. on Software Engineering, 7-45.