PROBLEM BASED STRATEGIES + ALGEBRAIC ABSTRACT DATA TYPES =
PROGRAMS

FRANCISCA LOSAVIO, ALFREDO MATTEO
Centro de Ingeniería de Software Y Sistemas ISYS
Escuela de Computación, Universidad Central de Venezuela
Ap. 47567, Los Chaguaramos 1041-A
Caracas, Venezuela

FRANÇOISE SCHLIE RGER
Laboratoire de Recherche en Informatique L.R.I.
Université Paris-Sud, 91405, Orsay Cedex, France

ABSTRACT - A methodology is presented for the assisted construction of programs, involving
decomposition of abstract data types which are formally specified as algebraic abstract data types. In order to
assure the construction of robust software, exceptional situations are taken into account very early in the
development process and they are also used to structure the corresponding program. As an example to
illustrate the method, we have selected to construct the program corresponding to the binary tree search
problem. Even if the methodology is sufficiently general to be used with any programming language
holding genericity, modularity and exception handling features, Ada has been selected as the target language
in this work. The program unit implementing the abstract data type, a decomposition scheme for
representing common algorithms on the data type structure and an appropriated construction strategy
depending on the nature of the problem, are the main elements used for program production. Finally, the
method is compared with a similar approach and the automation with respect to code generation is
evaluated.

INDEX TERMS - Assisted program construction, abstract data types, algebraic specification language,
construction strategies, decomposition schemes, exception handling, Ada language, software engineering.

1. INTRODUCTION.
A main concern of software engineering is the systematization and automation whenever possible of
software development. This work presents a methodology for program development [17], focusing on the
assisted program construction approach, where program generation is done with user intervention. The
program development is centered on the decomposition of algebraically specified abstract data types [9],
[10], based on the selection of operations defined on the type, called the interest type. Our approach to
program construction considers two aspects: a construction strategy related to the kind of problems and a
decomposition scheme on the data structures involved. Two levels of abstraction are taken into account, the
specification and implementation levels. A convenient language expresses the development in each level:
the PLUSS (a Proposition of a Language Usable for Structured Specifications) language [4] and a high level
programming language, Ada\textsuperscript{2} [5] in our particular case. We point out that even if we have used Ada as our
target language, the methodology is sufficiently general to be applied to any language holding modularity,
genericity and exception handling features. These characteristics must also be reflected into the specification
language. Moreover, we make extensive use of exception handling as a programming technique, enhancing
the consideration of exceptional situations at early stages of software development (exception confinement).
The technique used for decomposing the abstract data type allows to handle exceptions as soon as possible
(defensive programming). These two issues contribute to the construction of robust programs. Moreover,
the algebraic specification formalism used permits to write modular, hierarchical and parameterized
specifications, assuring the reusability of the software component produced. Several tools integrated into

\begin{footnotesize}
\begin{itemize}
\item[1] This work is partially supported by the Postgraduated Cooperation Program PCP-Informatics of the
CEFI-CO NICIT
\item[2] Ada is a trademark of the U.S. Government Adjoint Program Office.
\end{itemize}
\end{footnotesize}
the M-APEX prototype programming environment [15] support the methodology, allowing the assisted construction of Ada programs.

The paper is structured in four sections, besides this introduction and the conclusion. Section 2 constitutes our framework, describing the previous and related works, including the algebraic specification formalism used. Section 3 presents the basic concepts and notations involved, the decomposition scheme and construction strategies. Section 4 contains the process for program construction, with an example showing the development of an application, the binary tree search. Finally, in Section 5 an evaluation of the proposed methodology is presented, establishing a comparison with the example seen in Section 4 and a similar case taken form the works of Rich and Waters [18].

2. CONTEXT.
The antecedents of this work and the algebraic specification formalism on which it is based are presented in what follows.

2.1 Antecedents.
The present work is situated in the major field of automatic programming, where we can distinguish two main approaches:

1. The complete automatic program generation, with no user intervention: - The program synthesis approach [16]. Program synthesis systems have been built following these techniques: FSD [1], PROSPECTA [13] and CIP-S [2] - The automatic program generation and program construction selecting reusable modules, for limited applications, are techniques basically used in the Information Systems domain [14].

2. Program generation with the user intervention, with only partial automation, called assisted program construction; its antecedents are the works of Waters [21].

The proposed methodology for program construction, placed within the field of assisted programming, is based on Rich and Waters general notion of cliché [18], [21], constituted by a skeleton (present at each occurrence of the cliché), some roles (changing at each occurrence) and some constraints on the roles. Particularly, we are interested in algorithmic clichés. Its knowledge is represented by usual algorithms such as enumerating, searching and sorting on data structures. A recent work of Waters [22] suggests the inclusion of cliché-based edition of programs to programming environments, by introducing small changes in usual syntax-driven editors. The works of Gresse [8] and Losavio [17] are based on the common principle of abstract data type decomposition, where the data types are algebraically specified and data type decomposition means to consider the "algorithmic nature" of a data type, in the sense that the data structure is separated into different component objects by means of the operations defined on the structure. The templates or decomposition schemes reflect the way in which the operations are grouped (algorithm) in order to obtain the different objects of the type. Moreover, [17] deals with exception handling, following a defensive programming style. The common underlying idea is to take account of the fact that the data structure is strongly reflected into the structure of the program using those data [11]. Other interesting approaches that also consider the data translation are Jackson [12], Veloso et al. [19]. Volpano and Kieburz [20] use the concept of software template for specifying an algorithm defined by a sequence of recursive equations over the values of an abstract data type. The software template relates the abstract data type to its different implementations. The initial algorithm is given as a template, whose "instantiation" generates automatically the code corresponding to the implementation of the initial type. There is no transformation of the data in the sense of Gresse, Jackson and Veloso, followed by reduction and decomposition in order to solve a problem. The approach followed by Foisseau et al. [6] allows the description of programs at three different stages. Their main concern is an integrated and homogeneous presentation of the different languages employed at the different stages of the program development, giving
minor importance to the program construction methodology. Later, the earlier methods of Green and Losavio are generalized [15], introducing the Abstract Decomposition Scheme notion, independent from the particular data type and the use of parameterized algebraic specification. The decomposition scheme adapted to a particular data structure or type, called Concrete Decomposition Scheme, is obtained concretizing the abstract decomposition scheme.

2.2 Algebraic specifications.

Our methodology is founded on the algebraic specification formalism [7]. An algebraic specification [9], [10] defines a class of algebras or models. An algebra is just a possible implementation of the sorts and operation names which occur in the specification. A data type is characterized by one or more sets of values and by the operations performed on those values. It is defined as a many-sorted algebra, constituted by one or several sets of values (carriers), and some operations defined on these sets, which are named (sorts). The signature \( \Sigma \) of a data type is constituted by its sorts, the names and arity of its operations (domains and co-domains). Given a signature \( \Sigma \), the many-sorted algebra \( \Sigma \)-algebra, is an algebra where sets and operations are named following the names of \( \Sigma \). It corresponds to any implementation of the names of \( \Sigma \). A \( \Sigma \)-term is any valid composition of sorted variables and operations of \( \Sigma \). If the \( \Sigma \)-algebra is partial, the operations are partial ones. We deal here with algebras where the operations are total ones, called E,R-algebras (Error, Recovery algebras) [3], which are \( \Sigma \)-algebras where the carriers are divided into two disjoint sub-sets constituted by erroneous values and non-erroneous values. Exceptions and their eventual recovery are specified by means of two subsets of declarations. The E,R-signature is constituted by adding to \( \Sigma \) these two subsets. An abstract data type (ADT), is then a class of many-sorted algebras with the same signature and some specified common properties. In our framework, only finitely generated algebras w.r.t. the declared set of generators are considered. An algebraic abstract data type is the definition of an ADT by means of a signature and a set of axioms. In the case of E,R-algebras, the axioms are separated into ok-axioms and er-axioms. Only ok-axioms and exception declarations are used in this work. The presentation of an algebraic ADT is a pair \( \langle \Sigma, E \rangle \), where \( \Sigma \) is a signature and \( E \) a set of \( \Sigma \)-terms (\( \Sigma \)-axioms).

\[
\begin{align*}
\text{SPEC : } & \text{BIN_TREE [NODE]} \\
\text{USE : } & \text{BOOL} \\
\text{SORTS : } & \text{tree} \\
\text{GENERATED BY :} & \\
\text{empty :} & \rightarrow \text{tree} \\
\text{tree_gen :} & \text{node tree tree} \rightarrow \text{tree} \\
\text{OPERATIONS :} & \\
\text{root :} & \text{tree} \rightarrow \text{node} \\
\text{left_tree :} & \text{tree} \rightarrow \text{tree} \\
\text{right_tree :} & \text{tree} \rightarrow \text{tree} \\
\text{empty_tree :} & \text{tree} \rightarrow \text{bool} \\
\text{EXCEPTION CASES :} & \\
\text{no_root :} & \text{empty_tree(b) = true} \Rightarrow \text{root(b)} \\
\text{OK-AXIOMS :} & \\
\text{empty1 :} & \text{empty_tree(empty) = true} \\
\text{empty2 :} & \text{empty_tree(tree_gen(n,b1,b2)) = false} \\
\text{root1 :} & \text{root(tree_gen(n,b1,b2)) = n} \\
\text{left1 :} & \text{left_tree(empty) = empty} \\
\text{left2 :} & \text{left_tree(tree_gen(n,b1,b2)) = b1} \\
\text{right1 :} & \text{right_tree(empty) = empty} \\
\text{right2 :} & \text{right_tree(tree_gen(n,b1,b2)) = b2} \\
\text{WHERE :} & \\
\text{b,b1,b2 : tree} \\
\text{n : node} \\
\text{END BIN_TREE}
\end{align*}
\]

Figure 1. Algebraic specification of the binary tree data type.
The algebraic specification corresponding to the binary tree data type is shown above in Figure 1. Notice that the complete specification includes also the NODE and BOOL specifications, which are not given here, in order to facilitate the reading.

3. THE CONCEPTS.
In what follows, the basic notions and terminology underlying the proposed methodology are presented.

3.1 The decomposition scheme.
The intuitive notion of decomposition of a data type means to consider its data structure from an algorithmic point of view, separated into different parts constituted by a set of operations defined on the data type. Each one of these parts must contain at least one decomposition operation, that is to say, a destructor operation for the elimination of the objects of the structure. The algorithm is reflected in the way in which the operations constituting the different parts are placed in order to obtain these objects. As an example, let us consider the binary tree data type; in order to algorithmically deal with the tree, we may require to separate its root and process it accordingly, then recursively, we apply the same reasoning for the left and the right tree. Consequently, we consider this data structure divided into three objects: the root, the left tree and the right tree. An abstract decomposition scheme (ADS) [15], is a general formulation for this decomposition notion, allowing the expression of the algorithmic structure of a data type, by means of the operations defined on the type, which is algebraically specified. We will use the following notations: in a structured data type, TS denotes the type of the structure. The type of an element of the structure will be denoted by TE. The variables $S, S_1, \ldots, S_n$ denote objects of type TS and the variables $e_1, \ldots, e_m$ denote objects of type TE. An ADS is constituted by a header and a body. The header is composed by an object $S$ of type TS, called the input variable of the ADS, which is decomposed (denoted by the symbol "\rightarrow") into objects $e_1, \ldots, e_m$ of type TE and objects $S_1, \ldots, S_n$ of type TS, called the output variables of the ADS. The header of an ADS is denoted by: $\{S : TS \rightarrow e_1, \ldots, e_m : TE; \ S_1, \ldots, S_n : TS\}$. Let V1 and V2 denote the set of output variables and input variables respectively. The variables of types different from TE and TS, $X_1, \ldots, X_r, X_k$, with $k \in \mathbb{N}$ of types $T_1, \ldots, T_k$ respectively, denote the local variables. The operations defined on these types are called observers. For example, on the array data type, some computations on the array index are necessary to access part of the array structure. Set $Z_r$ denotes a subset of the variables in $X_r$ which may be empty if no local variables are present. The body of an ADS is constituted by a set of $r$ blocks, $1 \leq r \leq k$, and an exception handler for each block. In this presentation, those destructors, whose co-domain is type TE, will be called selectors. Those whose co-domain is type TS will continue to be called destructors. We distinguish two kinds of blocks, the selectors BLOCK_SEL and the destructors BLOCK_DES:

1. The body of BLOCK_SEL, is constituted by:
   - assignments of the local variables of set $X_1$, $1 \leq i \leq m$, by the set of observer operations OP_PREL. For ease of notation, the arguments of OP_PREL are not shown and the whole set of assignments to local variables are denoted by $X_1 \leftarrow OP_PREL$, where the symbol "\leftarrow" indicates an assignment.
   - assignments of an object $e_1$ of type TE by a selector operation SEL; the arguments of SEL are $S$ and some of the variables of $X_1$ denoted by $Z_1$. The expression corresponding to this assignment is then $e_1 \leftarrow SEL(S, Z_1)$.

2. The body of BLOCK_DES is composed by:
   - assignments of local variables as those mentioned for the first kind of block;
   - assignments of an object $S_j : S \in n$, of type TS by a destructor operation DES, as mentioned for the first kind of block.
The number of blocks of an ADS is then equal to the number of its output variables, \( m+n \). The exception handler is placed at the end of the block's body and contains exceptions \( \text{EX}_{\text{OP}_{\text{PREL}}_r} \) for \( 1 \leq r \leq k \), \( k \leq m+n \), raised by operations \( \text{OP}_{\text{PREL}}_r \) and exceptions \( \text{EX}_{\text{SEL}}_i \) or \( \text{EX}_{\text{DES}}_i \), raised by operations \( \text{SEL}_i \) or \( \text{DES}_i \), present in the body of \( \text{BLOCK}_{\text{SEL}}_i \) or \( \text{BLOCK}_{\text{DES}}_i \), respectively. The symbol \( \rightarrow \) denotes the raising of an exception by an operation. It is assumed, as we have already stated, that the expression \( \text{OP}_{\text{PREL}}_r \rightarrow \text{EX}_{\text{OP}_{\text{PREL}}_r} \) represents the set of exception handlers for \( \text{OP}_{\text{PREL}}_r \).

The complete ADS is given below on the left side of Figure 2; on the right side, a concretization of the ADS corresponding to the binary tree is given.

| {S:TS \( \rightarrow e_1, \ldots, e_m:TE; S_1, \ldots, S_n:TS \)} |
| \( X_1:T_1, \ldots, X_k:T_k \) |
| \( \begin{align*} |
| & X_1 \leftarrow \text{OP}_{\text{PREL}}_1, \\
| & e_1 \leftarrow \text{SEL}_1(S_1) \\
| & \text{exception} \\
| & \text{OP}_{\text{PREL}}_1 \rightarrow \text{EX}_{\text{OP}_{\text{PREL}}_1} \\
| & \text{SEL}_1 \rightarrow \text{EX}_{\text{SEL}}_1 \\
| & \text{end} \\
| \end{align*} \) |

\[
S: \text{BIN}\_\text{TREE} \rightarrow e_1: \text{NODE}; S_1, S_2: \text{BIN}\_\text{TREE}
\]

\[
\begin{align*} \\
\text{begin} \\
& e_1 \leftarrow \text{root}(S); \\
& \text{exception} \\
& \text{root} \rightarrow \text{no}\_\text{root} \\
\text{end} \\
\end{align*}
\]

\[
\begin{align*} \\
\text{begin} \\
& S_1 \leftarrow \text{left}\_\text{tree}(S); \\
\text{end} \\
\end{align*}
\]

\[
\begin{align*} \\
\text{begin} \\
& S_2 \leftarrow \text{right}\_\text{tree}(S); \\
\text{end} \\
\end{align*}
\]

\[
\begin{align*} \\
\text{begin} \\
& X_{m+1} \leftarrow \text{OP}_{\text{PREL}}_{m+1} \\
& S_{1} \leftarrow \text{DES}_{1}(S_{1}) \\
& \text{exception} \\
& \text{OP}_{\text{PREL}}_{m+1} \rightarrow \text{EX}_{\text{OP}_{\text{PREL}}_{m+1}} \\
& \text{DES}_{1} \rightarrow \text{EX}_{\text{DES}}_{1} \\
\text{end} \\
\end{align*}
\]

\[
\begin{align*} \\
\text{begin} \\
& X_{m+n} \leftarrow \text{OP}_{\text{PREL}}_{m+n} \\
& S_{n} \leftarrow \text{DES}_{n}(S_{n}) \\
& \text{exception} \\
& \text{OP}_{\text{PREL}}_{m+n} \rightarrow \text{EX}_{\text{OP}_{\text{PREL}}_{m+n}} \\
& \text{DES}_{n} \rightarrow \text{EX}_{\text{DES}}_{n} \\
\text{end} \\
\end{align*}
\]

**Figure 2.** Abstract and Concrete Decomposition Schemes.
3.2 The construction strategies.

The construction strategy is an algorithm which depends on the nature of the problem (for example counting or searching). It relates the problem we are building, called the Problem of Interest (PI) and the decomposition scheme, by appropriate combinations of the input and output variables of the problem and the output variables of the involved decomposition scheme. The construction strategies greatly contribute to structure the resulting algorithm, organizing the different blocks constituting the decomposition scheme. Several strategies [15] are present in our methodology: COMBINE, TEST, TEST-COMBINE, ESCAPE. Each strategy is oriented towards a particular kind of problem: COMBINE for enumerating, TEST and ESCAPE for searching, TEST-COMBINE for sorting. In order to ease the reading, we will be only concerned here with the TEST strategy, but a similar reasoning may be applied for the other strategies.

The TEST Strategy.

This strategy is used to solve search type problems. A case analysis is established over the elements of the data structure, in the respective selector block, which are compared with the element to be looked for. If the condition is verified, the result of the problem is returned, otherwise the destructor blocks are activated, in order to continue the search process on the rest of the data structure (e.g. array search). Conditions on intermediate results may also be established, in order to automatically determine in which part of the data structure the search process should be continued (e.g. binary tree search).

3.3 Combining construction strategies and decomposition schemes.

TEST is applied when the case analysis is performed on intermediate results computed in each block of the abstract decomposition scheme. The output variables of the decomposition scheme of set V1, computed in each selector block, and the input variables $I_{p},...I_{q}$ of PI of set V2, are used as arguments of the functions $F_{r1,...,rs}$, introduced for computing the intermediate results $R_{r}$. In the destructor blocks, the result of the recursive call to PI is used for establishing the conditions. An intermediate result $R_{r}1srsk$, is then computed in each block by means of a function $F_{r}$, and a condition $cond(R_{r})$ is established on the obtained result. If the condition is satisfied the result is returned by function $F_{r}$, else, the next block is activated, and so on. When the activated block is of kind BLOCK_DES, the intermediate result computed within the block is the output of the recursive call to PI.

Algorithm corresponding to the TEST strategy.

The general algorithm or program skeleton, which is independent from the data structure and corresponds to this strategy, is shown in Figure 3. In order to simplify the text, we present only one selector block and two destructor blocks. Notice that it is written in a target language style.
4. THE PROGRAM CONSTRUCTION PROCESS.

This section describes the process for constructing a program, presenting the information needed and the development of an application as an example.

4.1 Required information.

In order to construct a program using the proposed methodology, the following data is required: A Specification Library containing commonly used abstract data type (sequences, trees, etc.), written in a formal specification language, in our case, the PLUSS language. A Program Library with the implementations corresponding to the involved abstract data type algebraically specified. Since the Ada language is used, we speak of an Ada Library, containing the Ada packages implementing the operations of the abstract data type present in the Specification Library. A Strategy Library containing the program skeletons corresponding to different kinds of problems.

4.2 The process.

Step 1. Identify the kind of problem (enumerating, sorting or searching), which will be called the Problem of Interest (Pi), and select an available strategy according to this kind (e.g. a tree search). We select a program skeleton (e.g. see Figure 3) from the Strategy Library, which is independent from the data structure and the target language.

Step 2. Identify the data types involved in Pi. Select the interest data type (e.g. binary tree) which has to be decomposed, among the data structures of Pi

Step 3. Select the target language. Select the corresponding Program Library. An initial incomplete program, written in the target language, is thus obtained.

In our particular case, the operations on the algebraically specified data type (e.g. Figure 1) correspond to the operations on the Ada package (see in the Appendix the binary tree package) implementing the interest type, contained in the Ada Library.

Step 4. Complete the exception handlers with the required actions. During this step, which also involves renaming of the operations generated by the strategy, new sub-problems may be introduced.

Step 5. Solve the sub-problems generated in Step 4. These may be constructed applying again Steps 1 to 4, but they will generally consist of elementary operations. The process ends when all the generated sub-problems are completely implemented.

4.3 Example.

Suppose that the problem of interest Pi is the search of an element in a perfectly balanced tree with n nodes. The exit or failure of the search must be signaled. We are going to construct the program corresponding to Pi, using the proposed method.

Step 1. Pi is a kind of search problem, for which we select the program skeleton shown below, from the Strategy Library.

Step 2. The interest data type is a binary tree.

Step 3. The target language is Ada. The sp_bin_tree package is selected from the Ada Library. The program has to be completed with variable renaming and exception handlers.

Step 4. After having completed Step 3, the program shown in Figure 4 below, is obtained. Notice that there are no sub-problems, so the process ends.

Notice that the selector block returns the result value ‘true’ if the element is found. Otherwise, the first destructor block is responsible for searching on the left tree branch. If the element is not found there, the second destructor block performs the search on the right tree branch. If the search fails, the result value ‘false’ is returned by means of the exception no-root.
function Pi (S: TS; I<1, ..., I_q) return TTS
begin
  -- local variable declarations
  X_1 : T_1; X_2 : T_2; X_3 : T_3; S_1, S_2 : TS; c_1 : TE; R_1, R_2, R_3 : T;
  begin
  -- block BLOCK_SEL_1
    X_1 := OP_PREL_1
    c_1 := SEL_1 (S, Z_1)
    R_1 := F_1 (c_1, I_1, ..., I_q);
    if cond_1 (R_1) then return F_1 (R_1, I_1, ..., I_q);
    else
      begin
        -- block BLOCK_DES_1
          X_2 := OP_PREL_2
          S_1 := DES_1 (S, Z_2)
          R_2 := Pi (S_1, I_1, ..., I_q);
          -- recursive call to Pi
          if cond_2 (R_2) then return F_2 (R_2, I_1, ..., I_q);
          else
            begin
              -- block BLOCK_DES_2
                X_3 := OP_PREL_3
                S_2 := DES_2 (S, Z_3)
                R_3 := Pi (S_2, I_1, ..., I_q);
                -- recursive call to Pi
                if cond_3 (R_3) then return F_3 (R_3, I_1, ..., I_q);
                else return F_3 (R_3, I_1, ..., I_q);
                end
            end
          end if;
        exception
        when EX_OP_PREL_3 => return F_3, 1 (X_3, I_1, ..., I_q);
        when EX_DES_2 => return F_3, 2 (S_2, I_1, ..., I_q);
        when EX_F_3 => return F_3, 3 (R_3, I_1, ..., I_q);
        when EX_F_4 => return F_3, 4 (R_3, I_1, ..., I_q);
        end;
      end
    end if;
  exception
  when EX_OP_PREL_2 => return F_2, 1 (X_2, I_1, ..., I_q);
  when EX_DES_1 => return F_2, 2 (S_1, I_1, ..., I_q);
  when EX_F_2 => return F_2, 3 (R_2, I_1, ..., I_q);
  end;
end PI;
-- block BLOCK_SEL_1

Figure 3. Program skeleton for TEST, in a target language style.
5. EVALUATION OF THE METHODOLOGY.
The proposed method is evaluated in this section, comparing it with a similar approach and establishing the percentage of automation provided with respect to code generation.

5.1 Comparison with a similar approach.
Our methodology is mainly inspired on the works of Rich and Waters (see Section 2), so a comparison is established with their approach, even if other important works have been mentioned. The clichés in [18], are greatly influenced by the language and the plans (more general notions than the clichés) are based on the kind of problem, making use the data structures for constructing the corresponding algorithm very early in the process. Our program skeletons depend also on the kind of problem, but they are generated independently from the data structure, taking account of the abstract decomposition scheme notion. A particular data structure is concretized later in the process, with the target language choice (see Figure 5 below).
Figure 6, taken from [18], shows the plan for the binary tree walk. Notice that the traversal order must be determined at the moment of the language choice. In our case, the program is completely determined after applying the strategy (see Figure 3), during the initial step of the process. Moreover, in our approach, formal specifications and the accounting of exceptional situations play an important role, since they are involved at the specification stage of program development. The programs obtained following our methodology may then be claimed robust by construction.

![Diagram of the binary tree walk plan](image)

**Figure 6.** Rich and Waters' Plan for binary-tree-walk [18].

5.2 **Automation provided.**

The amount of code generated by this process depends on the number of selector and destructor blocks involved and on the construction strategy selected for solving a particular problem. As we have seen in 3.1, let \( k \), \( 1 \leq k \leq m + n \), be the total number of blocks present in the program skeleton, where \( m \) is the total number of selector blocks and \( n \) the total number of destructor blocks. For the TEST strategy (see Figures 3 and 4), the total number of automatically generated lines of code is computed as following:

\[
7(k-2) + 8 + 8 = 7k + 2
\]  

(1)

where \( 7 \) is the number of generated lines for each selector and destructor block, excepting the first selector block with 8 lines, including an additional exception handler for the returned function on the 'then' clause of the condition. The last destructor block, also with 8 lines, includes the exception handler corresponding to the 'return' on the 'else' clause of the condition (see Figure 3). The variable declarations, the function header and
the 'begin-end' clauses which have been automatically generated, have not been included in this computation, since they are not relevant to the result obtained. Now, for computing the number of lines that have to be manually completed by the user, we obtain:

\[ 4(k-2) + 5 + 6 = 4k + 3 \]  \hspace{1cm} (2)

where the right hand sides of the exception handlers and the 'return' clauses on the conditions have been counted. Considering a great value of k, from (1) and (2) we obtain that 57% of the code lines must be manually completed. For the remaining strategies, that are presented in [15], we proceed in a similar way and the percentage varies slightly, depending on the strategy.

6. CONCLUSION.

A general methodology for assisted program construction has been presented. It reposes on algebraic specifications of abstract data types, that are decomposed according to the operations defined on the type. Exception handling play an important role in the program development process, since exceptional situations are considered very early in the process (at specification level). Moreover, the algebraic specification formalism employed facilitates to express genericity and modularity, so robust and reusable code is assured, provided that the target language also holds exception handling, genericity and modularity features. The proposed method is mainly inspired on the works of Rich and Waters [18], [21] and its main difference is the independence of the data structure. Particularly, the general notion of abstract decomposition scheme, taken directly from the algebraic specification (independent from the implementation), assure that the derived algorithm is language independent; the language constructions appear at the moment of implementing it, during the last development stage. Moreover, the program skeleton is completely determined independently from the data structure, since it depends only on the problem to be solved, expressed by the corresponding construction strategy. With respect to the TEST strategy discussed here, 57% of the code lines generated as the program skeleton are completed with user intervention. However, these actions consist usually of elementary instructions, or the methodology is re-applied for solving more complex functions. Moreover, syntax driven edition within the supporting tools [15], which are not presented in this work, preserve the syntactical correctness of the program. The amount of automation of the method depends on the number of blocks present in the program skeleton and on the construction strategy used. Important issues to be considered in the near future are the enrichment of the strategy library with new variants, combining in different ways the input variables of the interest problem and the output variables of the decomposition scheme. Moreover, since much user intervention is required for completing the exception handlers, a classification of the exceptions could help to increase the automatic production of code.

7. REFERENCES.
APPENDIX.
The binary tree package of the Ada Library.
The binary tree data type is represented in a linked fashion. The implementation of the package specification part corresponds to the signature of the algebraic specification given in Figure 1.

```ada
-- Generic package for the binary tree data type. Notice that operation root raises an exception

generic
    type node is private;
package sp_bin_tree is
no_root : exception;
type tree is private;
function empty return tree;
function tree_gen (n:node; b1,b2:tree) return tree;
function root (t:tree) return node;
function left_tree (t: tree) return tree;
function right_tree (t: tree) return tree;
function empty_tree (t: tree) return boolean;
private
    type node struct; type tree is access node struct;
type node struct is
        record
            element : node;
            left : tree;
            right : tree;
            end record;
end sp_bin_tree;
```

Notice that the boolean type is predefined in Ada and that the node generic part corresponds to the parameter of SPEC BIN_TREE.