CAESAR: a system for CAse basEd SoftwAre Reuse

Gilles Fouquè and Stan Matwin
Ottawa Machine Learning Group / Department of Computer Science
University of Ottawa
Ottawa, Ontario K1N 6N5 Canada
(gfouque, stan)@csi.uottawa.ca

Abstract
This paper shows how the compositional software reuse approach can be fruitfully cast in the case-based reasoning (CBR) paradigm. Any CBR system in which the cases are software artifacts must rely on software-specific concepts to provide adequate knowledge representation. However, the fact that these software artifacts can be executed on a computer should yield stronger results than could be expected from generic CBR. The CAESAR system relies on advanced dataflow analysis concepts for its representation of knowledge about software modules. Cases and their fragments are retrieved and adapted to solve new problems. The new cases which result are executed on system-generated test sets to evaluate the results of CBR.

1. Introduction
Software reuse is generally understood as the reapplication of some of the various kinds of knowledge which were gathered during the development of one software system to the development of another. The expected gains in productivity and reliability from software reuse are particularly important in domains requiring large and complex software systems. Consequently, research in this area is sometimes perceived as "the challenge of the 90's in software development" [7][13]. Unfortunately even though reuse is a daily practice for many programmers, few tools promote and facilitate it. In the software engineering field most of the research into software reuse focuses on models of the development process conducive to reuse [4] rather than on the practicalities of reusing existing programs.

In this paper we present a machine learning approach to software reuse. Machine learning can be defined as "the reuse of previous experience to improve problem-solving" and as such is directly relevant to this task. Nevertheless, in spite of some interesting early work [12], the application of machine learning techniques to software reuse has received limited attention to date. For instance a flagship project in the area of AI-based software engineering tools, the (formerly) MIT Programmer's Apprentice [25], does not draw on techniques developed in machine learning. Newer work [5] does make use of the derivational analogy paradigm from machine learning however. According to this strategy, a new problem is solved by identifying a similar one that has already been resolved and addressing the differences between the two. This approach underlies our work, although we differ from Bhansali in believing that knowledge representation adequate for reuse must rely on software-specific concepts, rather than on a generic symbolic representation of cases.

The foremost practical technique for implementing software reuse is the compositional approach [22]. The compositional technique promotes reuse by giving the user access to a well-indexed and up-to-date library of existing programs. Prieto-Dim' original method promotes a multi-faceted categorization for indexing, and relies on classification and generalization of new cases by a skilled human librarian to ensure that the library grows in an orderly fashion.

Over the last decade issues of indexing and generalization have been thoroughly investigated in the machine learning field. During the same period software engineers developed advanced tools for the in-depth analysis of code to determine its basic components and the relationship which a program's flow of data and control establishes between them [16][24][30]. We combine these methodologies in a case-based software reuse architecture (CAse-basEd Software Reuse = CAESAR) which relies on code analysis tools for case representation. The next section presents the case-based reasoning approach and argues that it is conducive to solutions for the software reuse problem. We then describe CAESAR's individual components in more detail and illustrate the approach.
with an example synthesizing a new program from a set of specifications. The last section discusses future work in the context of limitations of the present system.

2. The Case-Based Reasoning approach

Cognitive science suggests that humans use their previous problem-solving experience to address a problem at hand (e.g. [10]). Experience is used not only positively to solve the problem, but also negatively to avoid previous failures in arriving at the solution. The Case-Based Reasoning (CBR) approach is based on this very idea: storing, retrieving and adapting cases of previously resolved problems in order to solve a new one. This approach is quite different from the inferential one where each solution is derived from scratch, relying on vast repertories of generic and domain knowledge. The steps involved in CBR can be summarized as follows [14]:
1. retrieve a case from the case memory,
2. compare the retrieved case to the current situation.
3. if necessary, adapt the current case in order to generate a case description that applies to the current situation,
4. use the previous case to generate inferences that can be transferred to help process the current input,
5. store in the base the results of the four previous processes.

The CBR paradigm has been applied to a variety of fields ranging from natural language understanding (e.g. [26], [23]), to planning [11], scheduling [29] and design [31]. Several implementations have established the viability of this approach in practice [21][11][15][6].

Notwithstanding this, none of the existing CBR systems has explored the analogy we propose, that many existing software libraries be considered as raw case-bases while program specifications are viewed as descriptions of planning goals. The work described here presents a design based on this analogy which defines a multi-layered, automatically-generated knowledge representation in which cases are programs and case-bases are program libraries.

3. CAESAR: general architecture

The goal of CAESAR is to help the user build a new program from its specification. Our approach to this task is to reuse a case-base of existing programs whenever possible in efforts to satisfy the specification. This process is adaptive, i.e. the system takes into account its previous experience in performing the task. Two aspects are particularly important:
1. CAESAR must include an adaptation process in order to modify, merge or extract fragments of code in the programs retrieved from its case-base,
2. CAESAR’s objective is not necessarily to fully satisfy the specification—it is not meant to be a program synthesis system—but to build a program which is useful to the user as a first draft.

The role of CAESAR’s user is thus not limited to providing the specification, but usually involves modifying programs built by the system. To facilitate the latter task CAESAR must provide some explanations about the programs it builds.

As shown in Fig. 1, CAESAR is based on the traditional CBR cycle of retrieval, adaptation and storage. One or more existing programs are retrieved from the case-base and combined into a new program which is stored back in the case-base, increasing its size. Another essential component of CAESAR is quite specific to software: the evaluation process. With case-based software reuse, the results of the retrieval and adaptation processes can be evaluated immediately and directly by the user ("the proof of the pudding is in the eating") using test generation techniques developed by the software testing community over the years [20].

![Fig 1: a global schema of CAESAR](image)

This schema consists of four processes:
a) RETRIEVAL: this process searches for fragments of code in the case base which satisfy part of the
specification expressed in the query. A code fragment can have associated with it constraints which the retrieval process must also satisfy. For example, a constraint can cause the retrieval of other fragments of code not explicitly mentioned in the specification but needed in order to satisfy input requirements of the first fragment.

b) ADAPTATION: often retrieved fragments of code must be altered and/or composed with one another to achieve the best possible match to the specification. The matching process is knowledge-based to ensure power, flexibility, and encapsulation of the necessary domain knowledge. The result of the adaptation is a set of new programs.

c) EVALUATION: the programs resulting from the adaptation process are run on test sets deduced from the program specification.

At the end of these three processes, the user has a static viewpoint on the new programs' conformance to the specification and a dynamic viewpoint on their behavior.

d) STORING: new programs are stored in the base together with the record of their success or failure to satisfy the specification as established by the evaluation process. This permits the constructive modification of the indexing scheme advocated by Aha [1] as the case base grows or its content shifts due to the changes in the external environment.

4. Knowledge representation

This section describes the representations needed to achieve the functionality in CAESAR described above. Two main structures dominate: a representation for each program and a representation for the whole case-base.

4.1. The case representation

An initial set of programs is stored in the case-base at the beginning of CAESAR's life. Those programs implement operations that are elementary (they do not perform their function with calling external subroutines) in the domain in question. Each program is a case and the whole set can be considered as the core case-base, i.e. the minimum knowledge needed to work in the application domain. To be good candidates for retrieval these initial cases must meet certain quality criteria. Those quality criteria essentially require that a program be structurally decomposable [9].

Each program is input as a set of instructions. Several transformations are then performed on a program to acquire the knowledge needed for the CBR. These transformations are well-known software-engineering techniques, often used to analyze code for purposes of modification [17], compiler optimization [2] or testing [20].

4.1.1. Specification of each case: Beside source code, a specification of each core program is the only other kind of information needed to initialize CAESAR. Syntactically a specification is a conjunctive formula in first order logic. This serves as a simplified description of a program's function, identifying its inputs and outputs, their properties (the characteristics of and restrictions on each input and output) and the relations between inputs and outputs (the behavior of the program). Part of those specifications are automatically extracted from the code. This description of a program in terms of five components is similar to the multifaceted approach to program specification [22] in which the relation is the action and inputs/outputs are the objects. We extend this methodology to include in the program representation, knowledge which is quite specific to software and which can be acquired automatically from cases.

4.1.2. The case-graph: Decomposition of a program into its basic blocks is the first of several steps to construct its case-graph. A case-graph is a representation of a program as a composition of different subparts, and of the relations existing between these subparts. The notion of a basic block [2] is commonly used in control and data flow analysis. Briefly, a basic block is a sequence of consecutive statements which may be entered only at the beginning and then is executed in sequence to its single end.

Three graphs are inferred from a program's basic block structure. These are the control flow graph, which describes the flow of control between basic blocks, the data flow graph, which describes the scope of each program variable in terms of basic blocks, and the precedence graph, which describes control flow relations constrained to be invariant by the flow of data. The precedence graph is deduced from the other two. These three graph structures produce a tree of blocks in which the program itself is the root and the basic blocks are the leaves. This hierarchy constitutes a program's case-graph and is stored in the case-base.

The quality criterion mentioned above permits the user to label some of the blocks with the name of the relation (function) they perform. This task is particularly important for blocks involving an output.

4.1.3. The cause-effect graph: A cause-effect graph [20] with layers for each kind of specification (five) can
be deduced from a program specification and its case-graph. This graph relates the inputs of the program to its outputs and is annotated with constraints describing the possible combinations of them. Then, a limited-entry decision table is constructed by methodically tracing back this graph. It is particularly useful to identify the high-yield set of test examples used in CAESAR’s evaluation phase.

On the basis of the constraints between basic blocks inherited from the case-graph, the cause-effect graph is converted into a limited-entry decision table in which the causes are block conditions and the effects are block actions. Each column of this decision table is then converted into test specifications and the user is invited to give an input/output example that satisfies the specifications. If a program successfully runs on each of these examples it is considered to have met the quality criteria.

4.2. Specification of the case-base

Thus each case is represented as a case-graph describing its functional decomposition and a cause-effect graph relating its input specification to its output specification. Effective retrieval of those cases which satisfy part or all of a query requires that they be indexed appropriately within the base. Indexing uses a slightly simplified representation of the whole base called the base-graph. (see example in sec. 5). The base-graph is produced by a composition of all cause-effect graphs in the base which merges common specifications and links each relation to the fragment of code which implements it (i.e. a node in a case-graph).

A domain-dependent hand-made rule base is added to the base graph to facilitate specification development and to increase the deductive capabilities of CAESAR so it retrieves more efficiently. This base consists of rules expressing relations between input specification, output specification, inputs, outputs or relations (cf § 5.2).

5. An example

CAESAR has been developed using libraries of numerical methods subroutines. This choice offers several advantages shared by most scientific libraries:

- **VOLUME**: their original purpose was to organize large numbers of routines,
- **QUALITY**: they were intended to provide reusable programs, so they meet the standards of programming quality which CAESAR needs,
- **TESTABILITY**: their routines can safely be reused as often as necessary,
- **AVAILABILITY**: they are readily available in the public domain or can be built from literature,
- **DEFINABILITY**: they employ a standard and well-known vocabulary which permit the user to express his problem in terms understood by CAESAR without difficulty,
- **DOCUMENTABILITY**: they have extensive and standardized documentation in the form of comments describing their functions.

The last point is rather important for the future development of CAESAR. The presence of comments in the code will allow us to build case specifications automatically by analyzing them along the lines suggested in Maarek et al [18] or Szpakowicz [28]. A specialized NL processor will extract relevant noun and verb phrases from the comments in a constrained subset of English.

In the remainder of this section, we will present an example of CAESAR’s retrieval and adaptation processes applied to a small sub-base of our base of matrix manipulation programs. Those programs find the solution of a system of linear equations in unknowns.

5.1. The case-graph

As illustration of the knowledge stored in the case-base for each program, we consider the program named GSHORT which solves linear equations in unknowns. GSHORT resolves the system:

$$
\begin{align*}
& a_{11}x_1 + \ldots + a_{1n}x_n = b_1 \\
& \vdots \\
& a_{nn}x_1 + \ldots + a_{nn}x_n = b_n
\end{align*}
$$

by Gauss elimination on matrices A and B in order to compute matrix X.

The code of the GSHORT program is:

```plaintext
for K := 1 to N-1 do
  for I := K+1 to N do
    begin
      M := -A(I,K) / A(K,K);
      for J := K+1 to N do
        A(I,J) := A(I,J) + M*A(K,J);
      end;
      B(I) := B(I) + M*B(K);
    end;
  X(N) := B(N) / A(N,N);
for I := N-1 to 1 step -1 do
  begin
    D := B(I);
    for J := I+1 to N do
      D := D - A(I,J) * X(J);
    X(I) := D / A(I,I);
  end;
```

93
The specification of this program is:

- **INPUTS**: matrix(A), matrix(B)
- **INPUT SPECIFICATION**: nb_line(A,N), nb_column(A,N), nb_line(B,N), vector(B)
- **OUTPUTS**: matrix(X)
- **OUTPUT SPECIFICATION**: nb_line(X,N), vector(X)
- **RELATION I/O**: gauss(A, X, B)

The case-graph of GSHORT is presented in figure 2 and its cause-effect graph in figure 3.

Because GSHORT is very simple, the case-graph is not very elaborate. In figure 2 on the left-hand side of the graph, the GSHORT instructions are represented without the control flow statements. The blocks C1, C2 and C5 are deduced from the control-graph. Those blocks are created first as they determine the control structure of the original program. Those control structures are supposed to compute macro-operations in order to achieve the final goal of the program. Then, the blocks C3, C4 and gauss are deduced from the precedence graph which indicates the interdependencies between blocks on the basis the variables they use. gauss(A, X, B) is the only node which the user must label, and it is the only relation appearing in the cause-effect graph in figure 3. In this graph, the input/output A, B and X are identified from the program while their specifications and types are defined by the user.

5.2. The base-graph

Let us suppose that six programs are currently stored in the base:

- **MATMUL**: computes the product of two matrices,
- **MATINV**: solves AX = B using basic matrix routines such as the matrix inverse,
- **GSHORT**: solves AX = B by gauss elimination (given above),
- **GPLUS**: solves AX = B by gauss elimination with interchanges,
- **GSYM**: solves AX = B for symmetric A,
- **JACOBI**: solves AX = B by Jacobi iteration.

The base-graph of this base is presented in figure 4.
An OR link, visible in figure 4 for \texttt{gauss(A, X, B)} means that there is more than one specification for the inputs or outputs of the same relation. A new input or output is defined for each specification. An underlined parameters identified the output of a relation when it is not obvious; \texttt{residual(A, X, B, &)} is the sole instance. In order to not overload the figure, the relations between the base-graph and the fragments of code have not been represented.

Three rules compose the knowledge needed for CBR matching of our example:

1. \texttt{R1 : nb\_column(A,N) and nb\_line(A,N) $\Rightarrow$ square(A)}
2. \texttt{R2 : nb\_column(A,1) and nb\_line(A,N) $\Rightarrow$ vector(A)}
3. \texttt{R3 : symmetric(A) $\Rightarrow$ square(A)}

The first two rules are rewrite rules; the last means that a symmetric matrix is a square matrix.

5.3. The query

Let us suppose that the user wants to resolve a linear system such as: \((A*B)^{-1}*X = C\). This problem can be specified as: \texttt{product(A, B, A1) and inverse(A1, A2) and gauss(A2, X, C)}

5.4. The solutions

The fragments of code involved in the computation of the fuzzy specification above are retrieved from five of the six programs in the case-base. They come from \texttt{MATMUL} for \texttt{product}, from \texttt{INVERT} for \texttt{inverse}, and from \texttt{GSHORT}, \texttt{GPLUS} and \texttt{GSYMM} for \texttt{gauss} (the relations retrieved have been italicized in Fig 4). These last programs are linked to the two cause-effect graphs of \texttt{gauss} (two graphs because of the OR link between the different specifications of \texttt{matrix(A)}). The fragments of code correspond to the Ci's shown in figure 1.

\texttt{CAESAR's} adaptive phase performs a number of operations on the matching blocks. To connect the output specification of \texttt{product} to the input specification of \texttt{inverse}, the rule R1 must be applied to the \texttt{inverse} inputs. Then the process matching the outputs of \texttt{product} to the inputs of \texttt{inverse} produces the binding \(M = P = N\), meaning that A and B are square matrices.

Precedence relations of the \texttt{MATMUL} program that computes the \texttt{product} specification indicate that this entire program is needed to compute the required specification. However, \texttt{CAESAR} is able to extract from the case-graph of the \texttt{INVERT} program only those
blocks involved in the relation inverse. It ignores the fragments of code involving the other matrix used by this program (which solve \( AX = B \) by computing \( A^{-1} \)).

Three programs have blocks which match the gauss relation; we will discuss each of these solutions individually. Three new programs will be created by combining the single fragments of code used in product and inverse with each of the three blocks involved in the gauss computation. These three programs correspond to the Pj’s of figure 1.

5.4.1 gauss relation implemented in GSHORT: The whole GSHORT program is involved in the computation of \( X \) from a square matrix \( A \) and a vector \( B \) (see fig 2). The adaptation process modifies the name of the variables of each fragments of the MATMUL, INVERT and GSHORT programs in order to put them together. The specification of the new program appears in figure 5. The succession of the three relations could be subsumed by a new relation \( \text{new-rel}(A, X, B) \) indicated by the parallelogram in figure 5.

A set of test cases can automatically be generated from this new cause-effect graph and from the case-base and executed on the new program.

5.4.2 gauss relation implemented in GPLUS: Using the matrix \( X \), GPLUS computes two other elements which are the residual part of the gauss resolution \( (R = B - A*X) \) and the determinant of \( A \) \( (D1 = \text{det}(A)) \). The code involved in the computation of \( X \) can be extracted from the GPLUS program without violating the precedence graph constraints shown in the case-graph. As with GSHORT, a new program is built. Its code is more complex for two reasons: a) GPLUS uses a more elaborate method to resolve the linear system (it interchanges the rows of the matrix \( A \)), b) statements involved in the computation of \( R \) and \( D1 \) are interleaved with those used to compute \( X \).

This case make clear the advantage of being able to execute a program. Although the first program is shorter and easier to understand (static point of view), the results computed by the second program are better (dynamic point of view). Execution gives the user better information with which to evaluate alternative programs.

The computation of \( R \) and \( D1 \) have been removed from the first GPLUS solution. GPLUS does not need other inputs to compute them, so a further solution which uses them could be suggested to the user. Figure 6 indicates the other specifications that would be involved.

5.4.3 gauss relation implemented in GSYMM: The constraint symmetric cannot be inferred directly from square and \( \text{nb-column} \), so the outputs of inverse do not fit the inputs of the gauss relation used in the GSYMM program. An exact match cannot be achieved with GSYMM. Nevertheless a program must still be constructed using the same process as before and submitted to the user for evaluation. Symmetric may be unimportant to the user’s application or he may edit the code to eliminate it, perhaps obtaining a better solution than the previous ones. The specification of the new program is presented on figure 7. The derived program fails on most of the test set which is derived from the relation obtained from the initial problem specification (Fig. 7).
Conclusion and future work

The design of CAESAR is based on the Prieto-Diaz faceted approach to software reuse - the well-known software reuse system successfully applied in practice. We have cast the faceted classification approach in the case-based framework. We have opted for an extended representation of programs as cases: one that exploits dynamic features of the program code to be reused, rather than the external, static facet descriptions. In CAESAR, the representation is automatically deduced from the programs to be reused, relying on well understood software analysis techniques. CAESAR is able not only to retrieve reusable components of programs satisfying a set of new specifications, but also to adapt those components into new programs. Moreover, CAESAR takes advantage of the runnable character of cases it operates on. The result of adaptation is tested on test data automatically derived from the specifications that CAESAR attempts to satisfy.

Implementation of the first version of CAESAR is under way at the time of writing.

There are several ways to improve and extend the design of CAESAR presented above. In this section we will discuss three of them: the organization of the case-base, the processes by which the new programs are built, and the storing process.

The first improvement concerns the development of the library. Currently CAESAR is a tool able to reuse existing libraries only and its success is dependent of the quality of its library. An interesting task would be to identify reusable programs in a specific domain in order to build new libraries meeting a certain quality criteria from the outset. CAESAR could then assist in the creation as well as the development of software libraries. For that purpose, an interesting starting point would be to use the slice approach [27] in order to retrieve the functional properties of a program.

The second point concerns the three steps (retrieval, adaptation and evaluation) involved in building a program. In the example presented above, all the code linked to any of the matched specifications is retrieved. It is then adapted to build one or more new programs, all of which are submitted to the user. This approach quickly becomes cumbersome with a case-base of realistic size; the user would give up when confronted with a large number of unordered new programs to verify. This shortcoming could be addressed by displaying the specifications of new programs after the retrieval process but before they are built. The user could quickly identify and reject incorrect or irrelevant ones. The advantages of this approach are twofold:
there are fewer different specifications than implementations (in the example, there are two different specifications and three resulting programs); and it is easier and quicker to evaluate an abstract specification than to verify an actual program. Moreover, after retrieval, the effort required to adapt a whole set of specifications to match the specification at hand can be computed [8] in order to provide more information to the user (it is during the adaptation process that some errors can raise in the new implementation).

The third improvement is by far the most ambitious. It is related to the storing process, which has not been detailed here other than to simply indicate that new programs which satisfy the specification and which have been approved by the user are inserted into the base. There are certain issues associated with case knowledge bases. The first is the utility problem [19]: the tradeoff between the usefulness of new cases and the increased time needed to retrieve cases from a larger library. It is not practical to store all the adapted programs in the base-base. A better solution is to store only those which are frequently reused. Consequently the case-base should record how often each of its new cases is reused. Frequently-used new programs may identify a composite relation which expresses a new concept which could usefully be articulated in the domain (e.g., new_ref in the example). Constructive induction [1] could be helpful in achieving this goal.

A program which has been explicitly rejected by the user provides CAESAR with another opportunity to learn. Such a program is a variant form of an acceptable program which shares part of its program specification but whose differences were deemed significant by the user. Exclusion links could be added in the case-graph to pinpoint the differences between the specification of the two programs as has been done in PROTOS [3].

ACKNOWLEDGMENTS
The work described here has been supported by grants from the Natural Sciences and Engineering Research Council of Canada, the Canada Centre for Remote Sensing and a grant Laboratoire des Affaires Etrangères. The authors thank Terry Copeck for his help in editing the paper.

REFERENCES

Academic Press eds.