# Tricky Errors in C Programs and Their Detection by Knowledge-Based Reverse Engineering

Stefan Trausan-Matu
Polytechnic University of Bucharest
**411** PUBLICATIONS **2,731** CITATIONS

Lorina Negreanu
Polytechnic University of Bucharest
**27** PUBLICATIONS **260** CITATIONS

# Tricky Errors in C Programs and Their Detection by Knowledge-Based Reverse Engineering

**Stefan Trausan-Matu**
**Lorina Negreanu**
**Mihaela Spalatelu**

"Politehnica" University of Bucharest
Computer Science Department
Splaiul Independentei, 313, Sector 6
Bucuresti, 77206

Romanian Academy Research Centre for
Machine Learning, Computational
Linguistics, and Conceptual Modelling
Calea 13 Septembrie, 13,
Bucuresti, 74311

ROMANIA
email:trausan@cs.pub.ro, trausan@valhalla.racai.ro, lorina@cs.pub.ro
http://www.racai.ro/~trausan

Presented at ROSYCS'98 Conference, "Alexandru Ioan Cuza" University of Iasi, Romania.

## 1. Introduction

Experienced programmers know a set of tricky errors to avoid. Such errors may not be detected at compile time. Some of them are even very hard to detect at execution time. Therefore, verification tools for detecting them are very useful. Unfortunately, such verification tools are very hard to develop because they imply either some kind of program understanding or, at least, of error pattern recognition.

One solution to the above problem is reverse engineering which means the following of the normal software engineering steps in a reverse order: A program, written in a programming language, is the source for several analysis and abstraction steps. The processing is directed towards obtaining a high level description of the program which may be a specification or an intermediate representation. This high level representation may be used for several aims like understanding or rewriting the program. A lot of research was done in the last years in reverse engineering of usual programming languages like C, Cobol, Fortran etc. [LHB91, Zim90].

A powerful reverse engineering system may include knowledge-based techniques to cope with the complexity of program semantics and with the needed abstraction steps. Knowledge-based reverse engineering systems enable the performance of some kind of, human-like program understanding [RiW90, Tra94a, Wil90, and Wil92]. Usually, such systems are built around a complex knowledge base of programming concepts and constructs [Ric85, SKW85, and RiW90].

In this paper is described a knowledge-based reverse engineering system for analysis of C program and for the generation of their knowledge-based representation. This representation may be used either for abstraction (using the abstraction facilities described in [Tra94a, Tra94b, and Tra94c]) or for the detection of complex errors, which cannot be discovered by usual C compilers. In this paper, the second case is taken into account. Therefore the system presented here finds complex errors in C programs.

The reverse engineering system for error detection consists of three main parts: a analyser of the C program which generates an intermediate code, a translator from the intermediate code into a knowledge based representation, and a collection of knowledge based routines for the detection of specific error patterns. The main advantage of the system is the knowledge-based representation, which allows powerful error checking. Another important advantage is the possibility to introduce new error cases to be tested due to the flexibility of the knowledge-based character.

The system was developed as a module for detecting complex errors in students' C programs. This module is a part of an intelligent tutoring system [TrN96]. Nevertheless, the system may be used on its own as a tool for the verification of C programs.

The organisation of the paper is as follows: The second section discusses several tricky errors that cannot be detected with current C compilers. The detection of this kind of errors is the goal of the system presented in this paper. The third section introduces the concept of knowledge-based systems and briefly presents the environment in which the system was built. The fourth section is dedicated to the presentation of the reverse engineering system. Some final remarks conclude the paper.

**Acknowledgements.** The research presented in this paper has been done partially as a research program funded by the Romanian Academy.

## 2. Tricky Situations at Execution Time of C Programs

In this section there are discussed several tricky situations in C programs. C compilers do not detect such errors. The next sections of the paper will introduce a way of detecting them by intelligent reverse engineering and error pattern recognition.

### 2.1. The Evaluation Order of Function Arguments

C gives the compiler the freedom to choose which arguments in a function to evaluate first; this freedom increases compiler efficiency but can cause trouble if, for example, we use an increment operator on an argument.

Let the following statements:

```
int n=5;

printf("%d\t%d\n", n, n*n++);
```

In this case, we want to print the number *n*, multiply it by itself to get the square, and then increase *n* by one. In fact, this program may even work on some systems - but not all. The problem is that when *printf*() retrieves the values for printing, it may evaluate the last argument first and increment *n* before getting to the other argument. Thus instead of printing:

```
5	25
```

it may print:

```
6	25.
```

### 2.2. Multidimensional Arrays as Parameters

Suppose we want to write a function to deal with two-dimensional arrays. We have several choices. We can use a function written for one-dimensional arrays on each sub-array. We can use the same function on the whole array as one-dimensional instead of two-dimensional. Or we can write a function that explicitly deals with two-dimensional arrays. The last case is the more interesting one.

For example, below is a function that takes a two-dimensional array address and the two array dimensions, number of rows and columns, and prints the array values.

```
void fmat (int **pm, int r, int c)
{  int i,j;
      for (i=0;i<r;i++)
            for(j=0;j<c;j++)
                  printf("%d\t",pm[i][j]);
      printf("\n");
}
```

Suppose we have this declaration:

```
int m[2][3] = {{1,2,3},{4,5,6}};
```

What happens when we call the function?

```
fmat(m,2,3);
```

The *m* being the name of an array is a pointer. As an array name *m* points to the first element of the array. In this case, the first element of *m* is itself an array of three *int*s, so *m* points to an array of three *int*s. Let's analyse the situation in terms of pointer properties:

- The value of a pointer is the address of the pointed-to-object. The name of an array is a pointer to its first element. Therefore, *m* equals to &*m*[0]. Next, *m*[0] is itself an array of three integers, so *m*[0] equals &*m*[0][0], which is the address of its first element, an *int*. In short, *m*[0] is the address of an *int*-sized object, and *m* is the address of a three-*int*-sized object. Because both the integer and the array of three integers begin at the same location, the *m* and *m*[0] pointers are the same numerically.
- Adding one to a pointer yields a value larger by the size of the pointed-to-object. In this respect, *m* and *m*[0] differ, for *m* points to an object with three *int*s in size, and *m*[0] points to an object one *int* in size. Therefore, *m* + 1 and *m*[0] + 1 are not the same numerically.

Individual elements using array and pointer notation may be represented as follows:

```
m[i][j] == *(*(m+i)+j)
```

The value *i* being the index associated with *m*, is added to *m*. The value *j* being the index associated with the sub-array *m*[*i*], is added to *m*[*i*], which is *(*m* + *i*) in pointer notation. Therefore, *(*m* + *i*) + *j* is the address of element *m*[*i*][*j*], and applying the operator * yields the contents at that address.

Now suppose we want to declare a pointer variable *pm* that is compatible with *m*, as parameter to a function. The type pointer-to-*int* won't suffice in this case. That type is compatible with *m*[0], which points to a single *int*, but we want *pm* to point to an array of *int*s. We can use:

```
int (*pz)[3];
```

which says that *pz* is a pointer to an array of three *int*s.

So, the declaration

```
int ** pm;
```

used in the function *fmat* is not correct, because:

```
pm[i][j] == *(*(pm+i)+j)
```

and *pm* points to an object with an *int pointer* in size, and not to an object with three *int*s in size.

## 2.3. The Function Return Value

One tricky error is to use the address of an automatic variable as the return value of a function. The automatic variables have local scope. An automatic variable comes into existence when the function that contains it is called. When the function finishes its task and returns control to its caller, the automatic variable disappears. The memory location (in fact the stack) can then be used for something else, so returning the address of that memory location is a mistake.

Let the following function definition:

```
char * funstr()
{
      char s[5] = "abc";
      return s;
}
```

If we want to print the content of the address returned by the function,

```
...
s1 = funstr();
printf("%s\n",s1);
...
```

the result won't be the expected one, because the content of the returned address is lost.

## 2.4. Function Type

Functions should be declared by type. A function with a return type should be declared to be the same type as the return value. The type declaration is part of the function definition. To use a function correctly, a program needs to know the function type. The compiler must have this information at hand before the function is used for the first time. One way to accomplish this task is to place the function definition ahead of its first use, although such placement may make the program harder to read. Also, the functions may be part of a library or in some other files. So we must inform the compiler about functions we use by declaring them in advance, otherwise the compiler will infer an *int* type.

For example, if the file M1 contains the function definition *fun*:

```
/* M1 */

double fun (double x)
{
      return x*2;
}
```

and in the file M2, we call the function *fun*, without declaring it:

```
/*  M2 */
double y = fun (2.0);
```

then *y* will have a wrong value, because the compiler doesn't know that *fun* has a *double* type, so it assume by default that it has an *int* value.

So the main concern is that the function declaration appear before the function is used:

```
/*  M2 */
extern double fun(double);
double y = fun (2.0);
```

## 2.5. Multiple Declaration

Some severe execution errors can appear if the same variable is declared as automatic and external and used in an inconsistent manner. For example, let the following declaration:

```
FILE * f;
```

In other function ( *main*(), for example), the same variable *f* is declared as automatic, and only there is properly initialised.:

```
void main()
{
        File * f;
        f = fopen(…..);
}
```

The global variable *f* is also used by other functions. But only in *main*() the variable *f* has as its right value a consistent memory address, in all the other cases its value is improper. The result will be a severe execution error.

## 2.6. The Need of Explicit Type Conversion

A typical execution error appears when we try to use the content of a variable declared as *void ** . Let the following declaration:

```
void * p;
……
(*p)++;
```

To increment the content of the variable *p*, the compiler needs to know the representation of (* *p*), which is unknown for a variable of type *void **. The solution is the explicit type conversion:

```
        int * a = (int *)p;
        (*a)++;
```

assuming an *int* pointer as example.

## 2.7. The Need of Initializations

A good programming rule is to initialise the variables when they are defined. Some severe errors can appear when not initialised variables of recursive types (such as trees or lists) are used. Let the following recursive type which defines the list data type:

```
typedef struct t_cel * List;

struct cel { int key;
                  List next;
};
```

The list-cell constructor is the function *cons*:

```
List cons(int k, List l)
{
        List aux;
        if ( (aux = malloc(sizeof(struct t_cel))) != NULL) {
                aux->key = k;
                aux->next = l;
        }
        return aux;
}
```

If there is intended to create and then use a list in function *main*():

```
main()
{
        List l;
        . . .
        l = cons(x, l);
        . . .
        while (l != NULL) {
        . . .
        l = l->next;
        }
}
```

a strange situation appears due to the *while* loop which will probably never finish, because the variable *l* was never initialised to NULL.

The solution is obvious :

```
        List l =NULL;
```

## 2.8. Out of Storage

The C language is a very nice and generous language. A lot of unconventional things may be done. Nevertheless, a lot of unexpected errors unsignaled at compilation time may appear. For example, one can write over the end of an array and no one will tell anything until the execution time. A very obscure error is to declare a variable as *char* and put a string (even with only one character) into it. Let the following code fragment:

```
void main()
{
        FILE * f; char c;
        ….
        f = fopen(“test”,”wt”);
        scanf(“%s”, &c);            /* we read only one character */
        fprintf(f, “%d”, c);
        fclose(f);
}
```

Apparently the code is correctly written. But after run it, the file *test* is empty. When a character is read with the *scanf* function using the “*%s*” specifier for strings (which is wrong, of course), the character will be stored at the left value of the *c* variable, and the Null character “\0”( the end of string marker) in the next memory location. But this memory location is not free. Because the local variables are stored on the stack, this location belongs to *f*. So, although *f* was correctly initialised, its content is now “\0” and the call to *fprintf(f, “%d”, c)* has no effect.

## 2.9. The Exit Branch in Recursive Functions

When writing a recursive function, one main concern must be the specification of the exit branch from the recursive successive calls, without which the process is endless. For example, if a tree with *int* keys is defined:

```
        typedef struct t_nod * Arb;
```

```
struct t_nod { int key;
               Arb ss, sd;
};
```

a recursive function for printing the tree keys may be:

```
void print(Arb a)
{
       print(a->ss);
       printf(“%d\t”, a-> key);
       print(a->sd);
}
```

Although the function seems correct, it will never end, because the exit condition is missing. The correct function has the following form:

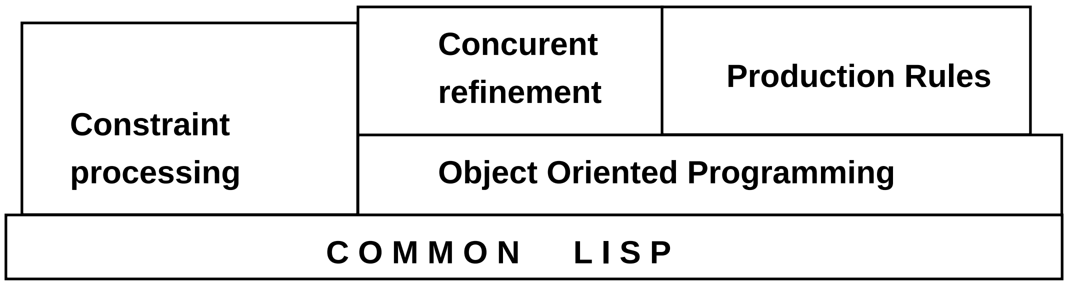
```
void print(Arb a)
{
       if (a != NULL) {
              print(a->ss);
              printf(“%d\t”, a-> key);
              print(a->sd);
       }
}
```

## 3. Knowledge-Based Systems; the XRL Structured Object-Oriented Knowledge Processing Environment

Knowledge-based systems (KBS) are programs in which a clear difference is kept between the knowledge they use and the procedures for processing it. This distinction permits the incremental development of the so-called knowledge bases, in which the knowledge is stored, while the processing procedures remain unchanged and are usually reused for a wide range of applications. This possibility is crucial for the development of computer programs for problems that are usually solved by humans which posses a large amount of knowledge in problem’s domain. The reason is, first, psychological: It is very hard for humans to describe the whole amount of knowledge they use. The description of knowledge is much easier in an incremental process. It is easier to understand what knowledge has the system by reading the knowledge base. In addition, the knowledge base may be used for several different purposes (i.e., constructing a solution, understanding a solution or generating explanations).

Knowledge may be represented in knowledge-based programs according to several paradigms: predicate logic, rules (“if” situation “then“ action), or frames (which have many common points with object-oriented programming). Some environments offer the possibility of the integration of several paradigms.

The system that is the subject of this paper has been developed in the classless object-oriented programming language XRL [BaT87a and BaT87b]. XRL is the substratum of a multiple-paradigm knowledge representation and processing environment (including rules, constraints, demons, and a blackboard-based concurrent refinement system). It was implemented in Common Lisp on several operating systems, from MS-DOS to VMS and UNIX. One of our main reasons for choosing XRL (vs., for example, CLOS) is its declarative, multiple-paradigm knowledge representation facilities and the multiple knowledge processing possibilities.



For feeling the flavour of the knowledge-representation language, three simple objects written in XRL are below exemplified. The "circle" object inherits the x and y slots from the "shape" object, has a particular slot named radius, and can respond to the draw message with a specific method (which overrides the method of shape). The circle32 object is a clone of the circle object (in a classless object-oriented programming language, there is not a cut between classes and instances; a particular object may be obtained by cloning an existing object):

```
(unit shape
  self (a meta-unit
         draw draw-shape)
  x undf
  y undf)

(unit circle
  self (a meta-unit
         supers (shape)
         draw draw-circle)
  radius undf)

(setq circle32 (a circle
                   x 2
                   y 3
                   radius 5))
```

An object may have a meta-object, which is declared, in the “self” slot. The “supers” sub-slot declares the list of objects from which the current object can inherit slots and methods. The assignment of methods to messages for the current object is described as pairs in the self slot.

Sending messages is accomplished with “msg”. For example, sending the draw message to it does the drawing of circle32:

```
(msg ’draw  ’circle32)
```

In XRL, any slot may have a value, which is a clone, like in the following example:

```
(unit map
  self (a unit supers (functional_on_list))
  name map
  body (a iterate_on_list
         c2c (a cons 1st (a funcall name f)
                     rest (a reccall name map args (f l)))))
```

An important function of the XRL language is “isa”, which tests whether an object has (or is an instance of an object which has) as ancestor (by the inheritance relation) another object. For example, the following test returns true if function f12 is a map function:

```
      (isa ‘f12 ‘map)
```

# 4. The Reverse Engineering System

The reverse engineering system for error detection consists of three main parts:

• a lexical, syntactic, and semantic analyser whose result is an intermediate code. This module is written using the classic Yacc and Lex programs.

• a translator from the intermediate code into a knowledge based representation. This module is written in the XRL language [BaT87a, BaT87b]. XRL is not only a language. It is an object-oriented development environment for knowledge-based applications.

• a collection of knowledge based routines for the detection of specific errors. This module can be considered as an expert system for error detection. Therefore it may be further extended with new knowledge.

The last two modules are built around a structured (frame-like [FiK85]) object-oriented knowledge base developed in XRL. This knowledge base includes a large set of programming concepts. It is the result of several years of research in reverse engineering, intelligent tutoring systems for programming, and in the theory algorithms and data structures [Tra94, TrN96, Tra97].

The result of the analysis part is intermediate code generation. A C source file is processed and two other files are produced, one containing the intermediate code representation (.grh) and the other the symbol table (.syt). The intermediate representation and the symbol table description are both LISP forms. Every intermediate form is a list. The head of the list is the operation code, and its second element is the operation internal code. The other elements of the list are proper to every operation type. Every form in the symbol table description is a list to. The list head is the variable or type identifier. The second element is the type or variable internal name. The other elements are specific.

The Lex and Yacc compilers were used to specify the lexical and syntactic analysers.

For example, starting from the following fragment of a C program:

```
{
int i;
. . .
for (i=0;i>10;i++)printf("\ni=",i);
. . .
}
```

There will be generated the following intermediate code:

```
(
 . . .
(ASSIGN id10 id8 id9)
(CONST id12 10)
(GT id13 id8 id12)
(POSTICR id15 id8)
(STRING id17 "\ni=")
(FUNCALL id18 id0 id17 id8 )
(FOR id20 id10 id13 id15 id18)
(BLOC id21 id20)
 . . .
)
```

and the following description for the “i” variable:

```
(
 . . .
(i id8 id21 int 0)
 . . .
)
```

The two files generated by the first part of the system are the input of the second part. During this second phase, a knowledge-based, object-oriented representation is generated. For each data element and for each programming construct, a clone of an object in the knowledge base is generated. For example, for the variable “i”, the following object is generated:

```
(a INT
    SCOPE  ID21
    NAME  I
    DEFINED-IN . . .)
```

For the POSTICR statement, which is the correspondent of i++, will be further translated from the intermediate code to the following node:

```
*** POSTICR *** is a POSTICR
  description: (POSTICR ID15 ID8)
  input vars: (ID8)  -- output vars: ID8
  uses: (ID8)
   ----ctrlflow in:GT
   ----ctrlflow out:FUNCALL
   ----data flow in:
   ----data flow out:
   in FOR
```

This node is a XRL clone of the posticr object of the knowledge base:

```
(unit posticr self (a unit supers (inc)))
```

Posticr inherits from inc:

```
(unit inc
;;;                           v=v+1
 self (a procAbst supers (selfAssign) genC genCinc))

 . . .
```

and so on. Below is a part of the taxonomy of objects for programming concepts in the knowledge base:

```
PROGRCONCEPT
      DATADESCR
            DATAELEM
                  CONST
                  VAR
                        INT
                        . . .
            DATATYPE
            . . .
      PRGND
            PROCDESCR
            EXPR
                  . . .
            STM
                  SMPL-STM
                        CONTINUE
                        RETURN
                        BREAK
                        ASSIGN
                              SELFASSIGN
                                    INDEXEDSELFASSIGN
                                    INC
                                          PREICR
                                          POSTICR
                                    . . .
                              . . .
                        . . .
                  CMPD-STM
                  . . .
            BLOCK
            . . .
```

The knowledge-based routines for error checking are rules, which has as a condition part an error pattern and as the action part some signaling code. For example, a fragment of a rule that tests if an array is returned is the following:

```
(if (and (isa x ’return)
         (isa v ’array)
            . . .)
  (format t "~%~%Error - ~A returns ~A, which is locally defined !~%"
      (fslot ’name (function_that_contains x))
      (fslot ’name v)))
```

The third part of the system is an extensible collection of such rules.

## 5. Conclusions

The system introduced in this paper is an extension of an intelligent reverse engineering system for translating FORTRAN programs to C programs called “Revenge” [Tra94a, Tra94b, and Tra94c]. In addition to that system a C analyser was developed and a collection of error rules were introduced. The system presented here may use all the abstraction or re-generating code (in C or pseudo-code) facilities of “Revenge”.

The error detection system was tested on several error cases from that presented in section 2. The successful results proved the viability of the approach. An important fact to mention is also the high speed of the testing.

The system will be used for testing students’ programs in the Computer Department of the Politehnica University in Bucharest. In fact, the system was designed as a part of a complex intelligent tutoring system developed by the first two authors [TrN96].

## References

[BaT87a] Barbuceanu, M., Trausan-Matu, St., XRL: An evolutionary multi-paradigm environment for AI programming, in Ph. Jorrand si V. Sgurev (eds.), Artificial Intelligence II: Methodologies, Systems, and Applications, North Holland, 1987, pag. 197-205.

[BaT87b] Barbuceanu, M., Trausan-Matu, St., Integrating declarative knowledge programming styles and tools in a structured object environment, in J. Mc.Dermott (ed.) Proceedings of 10-th International Joint Conference on Artificial Intelligence IJCAI’87, Milano, Italia, Morgan Kaufmann Publishers, Inc., 1987.

[FiK85] Fikes, R., Kehler, T., The role of frame-based representation in reasoning, Communications of the ACM, Vol.28, No.9, (sept. 1985), pp. 904-920.

[LHB91] Lano, K., Breuer, P.T., Haughton, H., Reverse-Engineering and Validating COBOL via Formal Methods, ESPRIT IPSS Results and Progress, 1991, pp. 284-305.

[Ric85] Rich, C., The Layered Architecture of a System for Reasoning about Programs, Proceedings IJCAI’85, pp.540-548.

[RiW90] Rich, C., Waters, R.C., The Programmer’s Apprentice, ACM Press, Addison-Wesley, 1990.
[Sch90] Schindler, M., Computer-aided software design. Build quality software with CASE, John Wiley & Sons, 1990;

[SKW85] Smith, D. R., Reasearch on Knowledge-Based Software Environments at Kestrel Institute, IEEE Transactions on Software Engineering, vol. SE-11, nr. 11, nov. 1985, pp. 1278-1295.

[Tra94a] Trausan-Matu, St., Intelligent Reverse Engineering, PhD Thesis, “Politehnica” University of Bucharest, 1994

[Tra94b] Trausan-Matu, St., An Evolutionary Knowledge-Based Framework for Reverse and Forward Engineering, in Studies in Informatics and Control, vol.3, no.4, dec.1994, pp. 309-323

[Tra94c] Trausan-Matu, St., Sistem bazat pe cunostinte pentru ingineria inversa a programelor, in Rev. Romana de Informatica si Automatica, vol.4, nr.1, 1994, pp. 21-35 (in Romanian)

[TrN96] Trausan-Matu, St., Negreanu, L., Sistem inteligent de asistare a instruirii, raport RR-14, Academia Romana, CCAIAPLNMC, iunie 96.

[Tra97] Trausan-Matu, St., Knowledge-Based, Automatic Generation of Educational Web Pages, in Proceedings of Internet as a Vehicle for Teaching Workshop, Ilieni, iun. 1997, pp.141-148.

[Wil90] Wills, L.M., Automated Program Recognition: A Feasibility Demonstration, Artificial Intelligence, 45 (1990), pp. 113-171.

[Wil92] Wills, L.M., Automated Program Recognition by Graph Parsing, AI-TR 1358, MIT, 1992.

[Zim90] Zimmer, J.A., Restructuring for Style, Software-Practice and Experience, vol. 20, no. 4, pp. 365-389, apr. 1990.