IMPLEMENTING A PL/M-TO-C CONVERTER WITH TaLE

Maarit Niittymäki

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1994-4

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1994-4, FEBRUARY 1994

IMPLEMENTING A PL/M-TO-C CONVERTER WITH TaLE

Maarit Niittymäki

University of Tampere
Department of Computer Science
P.O. Box 607
FIN-33101 Tampere, Finland

ISBN 951-44-3522-2
ISSN 0783-6910

Implementing a PL/M-to-C converter with TaLE

Maarit Niittymäki
Department of Computer Science, University of Tampere
Box 607, 33101 Tampere, Finland
email: csnima@cs.uta.fi

Abstract

A source-to-source converter translating programs from PL/M to C is described. The converter has been implemented with the language implementation system TaLE. Various problems specific to this converter are discussed. The problems are related partly with the particular language pair and partly with the object-oriented implementation tool.

1. Introduction

In some situations it is important to be able to translate existing programs written in one language to another language. We can be lacking of compiler of one language in a certain computer. Some programming languages may become obsolete leaving programs written in such a language without support. An alternative to connect programs written in different languages is to translate these programs to the same language.

In this report we describe a source-to-source translation from PL/M to C. There are several conversions of that kind, but our translator is rather different from the others, due to the implementation tool, TaLE (Tampere Language Editor) [JäKo93a, JäKo93b]. TaLE is a tool supporting the development of language implementation software in an object-oriented programming environment. It emphasizes software engineering qualities rather than contributions in formal languages, and this makes the system in many ways different from more traditional language implementation systems.

TaLE is written in an object-oriented language, Eiffel [Meye88]. In this report we assume that reader is aware of the concepts of the object-oriented paradigm.

A main principle in designing TaLE has been reuse. In language implementation this aspect has been mostly ignored, because programming languages have been regarded as indivisible entities. However, in languages common parts can be found, these parts are for example expressions, list structures, keywords and identifiers. In TaLE the reuse of language structures has been implemented in three ways. First, different language structures have been implemented in different classes according to object-oriented paradigm allowing the free combination of language implementation components representing various language features in different languages. Second, general structures can be implemented in abstract level, and each language implementor can refine them more precisely for his or her own purposes. Third, standard concepts and their implementations are built in the system, which enables language implementors to exploit them easily.

When using TaLE, language implementor defines a class for each nonterminal in the grammar. The components of the nonterminal can be given with a graphical palette. The components of the nonterminal can be nonterminals themselves. In the generated class there is an attribute for each nonterminal component of the structure; the name of the attribute is composed automatically by the system (class name plus a running number).

TaLE provides automatic parsing according to given grammar: we need not write any parsing code for the PL/M-to-C converter. TaLE is an open system allowing the user to specify language features using attributes and operations written in Eiffel. TaLE generates the parsing code in Eiffel, so the user-given features also have to be written in Eiffel, because they are merged with the parsing code. To implement the PL/M-to-C converter we have to give the PL/M grammar in the TaLE form and write an appropriate C-compiling routine for each nonterminal. Each C-compiling routine contains only calls of the C-compiling routines of the nonterminal components or simple printing commands. Because of that underlying idea of TaLE the C-compiling routines – like the nonterminal classes of the language – are very independent of each other.

Presentation proceeds as follows. In the second chapter we will introduce other existing source-to-source converters and compare them with our system. In the third chapter we will present some problems which are unrelated to PL/M and/or C, but which are characteristic to source-to-source translations. In the forth chapter we will discuss problems caused by the languages PL/M and C, and in the fifth chapter problems caused by the implementation tool. Finally, we conclude with remaks concerning the translation work in general and especially with TaLE.

2. Related work

Source-to-source conversions have been described in the literature. Many of them have Fortran as their source language [Frea81, SlWa83, PBBC88] and Ada as their target language [AGGH80, SlWa83, Mart86, PBBC88, MoWa91].

Some general problems can be recognized in source-to-source conversion. One of these, due to the different structures of source and target languages, is described in [MoWa91]. Certain structures of the source language have no counterparts in the target language, and in consequence converted programs may be clumsy. On the other hand some structures of the target language have no counterparts in the source language. As a result, some structures of the target language may remain without use and converted programs do not always resemble typical programs of that language. Accordingly, we can set at least two kinds of conflicting requirements to source-to-source converter. On one hand we can require that converted programs must be able to be compiled and act exactly like the original program, even though the structure of the program may be complicated and clumsy. That kind of programs may be difficult to maintain. On the other hand we can require that converted programs must be easy to maintain. In this approach we may even leave some structures without translating and add the translation afterwards by hand.

In the design and implementation of most of the existing converters the second requirement has been followed. In the current version of the PL/M-to-C converter we mainly use the second way, too. In addition, we have planned a solution which would be some kind of compromise of these two methods. We propose an interactive converter which asks the user for an appropriate conversion in different situations. The proposed method would offer some advantages: it reduces the amount of the code that remains hand written and the output text resembles more likely the program text of the target language.

In the design and implementation of the PL/M-C -converter we have met many of the problems described in the literature. On the other hand, many of our problems are different from other systems' problems because of the different programming paradigm. The other systems have mostly used traditional procedural languages in implementing their convertor, while we have used an object-oriented language. Many of the other systems have not applied an automated tool, whereas we have used an object-oriented language implementation system, TaLE.

A basis of the translation has been Mikkonen's study [Mikk91], where she has mapped the differences between PL/M and C and given some translation proposals both to hand written and automatic translation.

3. General problems

In this chapter we introduce some problems which are general to source-to-source converters but not to traditional compilers. These problems are caused by the fact that the target language is a high-level language which will be maintained.

Comments

When designing and implementing a compiler, comments in input program text do not cause any problems. They can be discarded, because the output program text is not meant to be read or updated. In the case of source-to-source converters comments should be transmitted to output program into appropriate places instead of discarding them.

Usually comments can be handled by storing them in the nodes of the parse tree that they refer to [SlWa83]. We have an analogical method in PL/M-C converter. In every class generated by TaLE there is an attribute `comment_list`, which is a list of comment strings. The scanner object of TaLE has an attribute `last_struct`, which is the latest structure created. When a structure has been created it puts itself as the latest structure to the scanner. Each time we find a comment in the input text, we read it to a string and connect the string to the `comment_list` of `last_struct`. After performing C-compiling routine to an object, we also perform comment writing routine, which prints out the `comment_list` if the object has one.

Compiler directives

Like comments, also compiler directives can be located anywhere in the program text. Hence, we handle compiler directive in the same way as comments, we even use the same list of strings to both comments and compiler directives. In addition, we have an object called `comment_handler` which handles both comments and compiler directives. However, there is at least one difference between comments and compiler directives: comments need not be analyzed but merely transmitted whereas compiler directives must be changed to their counterparts in C language. Because of that difference the object `comment_handler` has a comment writing routine which first checks if each string in the `comment_list` is a comment or a compiler directive. In the case of compiler directive, the string must be examined more accurately. First of all the directive string has to be analyzed to clarify which compiler directive is in question and what are the parameters of the directive.

File inclusion

A special case of compiler directives is include-directive used for including other files to the program file. However, the included files are written in the source language, and we cannot include the same files to the output program. We made the assumption that the user always translates the whole software, although one file at a time. If we do not change the names of the program files being converted, except the suffix part, which we transform from `.p` to `.c`, we can assume that every PL/M file appearing in an include directive always has corresponding C file, expect library files. So, translating an include directive we can copy the name of included file and change only the suffix part.

Macros

One problem specific to source-to-source converters is macros and their expanding. Many languages provide macros or other features for textual replacement. That kind of feature is at least in PL/M and in C; we will call this feature macro, even if it has a different name in PL/M. Since macros can change the syntax of the language, it would be very difficult to expand macros during parsing. So, we have to expand the macros before parsing and translation.

With macros one can also define constants or new types from existing types. If we expand all the macros, we lose some useful names which make the program more readable. Consequently, we have divided the items defined by macros into two groups: those we must expand and those we intend to preserve. In the first group we have macros whose extensions are or contain reserved words except type names. In the second group we have macros whose extensions are expressions or types. We have made a selective preprosessor, which expands only those macros belonging to the first group. When we will in future develop the PL/M-C converter, the user will be able to select the members to each group.

4. Problems caused by the language pair: PL/M and C

In this chapter we deal with problems caused by differences between the source language PL/M and the target language C. In the first section we introduce differences between the order of syntactic structures. In this section both languages have the corresponding components in their grammars, but these components are situated in different places. In the second section we introduce such structures which are not present in the target language at all.

4.1. Problems related to the order of language expressions

Example 1. Variable declaration

The components of the grammar structures can be in different order or the components can be divided into different structures in the source and target languages. Sometimes this difference is so small that it does not cause any severe problems. For example in the variable declaration of PL/M we first declare the name of the variable and after that the type of the variable. In C variable declaration has the reverse order. In this case an object of class `Variable_declaration` has the attributes `Variable_name_1` and `Variable_type_1`. The only thing `Variable_declaration` has to do in its C-compiling routine is to perform the C-compiling routines of its component structures in the order corresponding to C language. This can be stated as follows:

```
-- Variable_declaration

c_compile is
      do
            Variable_type_1.c_compile;
            Variable_name_1.c_compile;
      end; -- c_compile
```

However, there are many other cases where the differences in grammar structures are larger. Changing only the order of the nonterminal components is not enough if corresponding components do not appear in the same structures. For example, variable attributes are divided in a different manner in PL/M and C. In PL/M we have following rules in simplified form:

```
Variable_declaration –> Variable_name [Array_specifier]
                        Variable_type [Variable_attribute]
Variable_attribute > Public_attribute | External_attribute
Public_attribute –> [“PUBLIC”] [Locator] [Initialization]
External_attribute –> [“EXTERNAL”] [Initialization]
```

In the above grammar we have used the following notation. Optional components of the rule are within brackets. In the second rule of the example we have used a different production symbol (>). The explanation is that the nonterminals in TaLE are divided into structural and conceptual nonterminals. Structural nonterminals define an aggregation, and we use symbol (–>), while conceptual nonterminals define a subclass relationship, and we use symbol (>). Note that if there is only one component in the right side of the structure, we can distinguish the type of the nonterminal merely on the basis of the symbol.

When we have to define the same routine both in a superclass and in its subclasses, we have two possible approaches. One alternative is to use deferred routines in the superclass and not to define them until in the subclasses, while the other is to define a routine in the superclass as empty and redefine it in the sublasses. We have chosen the latter approach because in some situations we can take advantage of empty routines also in the subclasses without defining them at all. In addition TaLE supports this approach better.

The previous grammar shows that in PL/M variable attributes are always at the end of the declaration. Variable attributes can be among other things memory specifiers (“EXTERNAL” or “PUBLIC”) or variable initializations. The difference between those two languages is that in C memory specifications must be at the beginning of the structure and initializations at the end of the structure. However, we can solve the problem in an elegant way as follows. We divide the C-compiling routine of class `Variable_attribute` into two parts: one takes care of the C-compiling of memory specifiers and the other perfoms the C-compiling of initializations. In the subclasses of `Variable_attribute` they are splitted in corresponding way. For example class `External_attribute` has the following operations:

```
-- External_attribute

c_compile1 is
      do
            output.putstring(“extern “);
      end; -- c_compile1

c_compile2 is
      do
            Initialization_1.c_compile;
      end; -- c_compile2
```

Further, an object of class `Variable_declaration` calls both C-compiling routines of object `Variable_attribute`. The C-compiling routine is as follows:

```
-- Variable_declaration

c_compile is
      do
            Variable_attribute_1.c_compile1;
            Variable_type_1.c_compile;
            Variable_name_1.c_compile;
            Variable_attribute_1.c_compile2;
      end; -- c_compile
```

Another possible way to solve the above problem is to define only one C-compiling routine in class `Variable_attribute` and pass the information whether to C-compile memory specifications or initializations as a parameter. This solution has the disadvantage that C-compiling routines in different classes would have different parameters.

Example 2. Declarations

In the previous example we used several C-compiling routines in order to divide the corresponding C code in different places in output file. We can use several C-compiling routines for other purposes, too. In the previous case we performed both C-compiling routines in every case, but we can present C-compiling routines in such a way that they will be performed selectively in different cases.

The grammar of declarations in PL/M is constructed in such a way that in TaLE we can use class `Declaration` with two subclasses: `Declare_statement` and `Procedure_definition`. This means that a list of declarations can contain both variable declarations and procedure definitions mixed. In PL/M for example a procedure has a list of declarations followed by a list of statements. The first list can contain variable declarations and definitions of other procedures mixed with each other: nested procedures are allowed in PL/M, unlike in C. Consequently, we should be able to detach variable declarations and procedure definitions from each other, because in the corresponding C code we can preserve variable declarations inside the procedure but definitions of nested procedures should be moved outside the procedure.

In order to distinguish the C-compiling routines of variable declarations and procedure definitions from each other, we can use several C-compiling routines combined with late binding. In the class `Declaration` we have two C-compiling routines, which both are empty. Then in class `Variable_declaration` we redefine one (say first) of those routines by defining how variable declarations should be translated. Further, in class `Procedure_definition` we redefine the other (second) of those routines by defining how procedure definitions should be translated. This can be seen below.

```
-- Declaration

c_compile1 is
-- compiles declare statements
      do
      end; -- c_compile1

c_compile2 is
```

```
      -- compiles procedure definitions
            do
            end; -- c_compile2
```

In class `Declare_statement` we redefine the first C-compiling routine:

```
      -- Declare_statement

      c_compile1 is
      -- compiles declare statements
            do
                  -- necessary statements
            end; -- c_compile1
```

In class `Procedure_definition` we redefine the second C-compiling routine:

```
      -- Procedure_definition

      c_compile2 is
      -- compiles procedure definitions
            do
                  -- necessary statements
            end; -- c_compile2
```

When we have a list of declarations, we perform the first C-compiling routine for every component in the list. If the subclass of a list component is a variable declaration, necessary C code is written to the output file. But if the subclass of a list component is a procedure definition, nothing happens. When we perform the second C-compiling routine to every component of the declaration list, the effect is the opposite. In this way we manage to distinguish the items of a list without knowing the subclass of the list item.

Example 3. Procedure parameters

Formal parameters are declared differently in PL/M and in C. In PL/M the names of the formal parameters are listed within parentheses in the header of a procedure. Inside the procedure formal parameters are declared with their types among the other variables of the procedure. In other words, the order of the parameters is given in the procedure header and the type of each parameter can be found in variable declaration besides the types of other variables. Produced C code is meant to be according to ANSI standard, so both the names and types of formal parameters should be given within parentheses in the procedure header, and the other variables should be declared separately from formal parameters.

In both languages we have the same information of the parameters, although the information is in different places. We can apply the following algorithm to search for the necessary information in PL/M code and transmit it to appropriate places in C code. We go through the list of formal parameters and for each parameter we search for the same name from the declarations. When we find a formal parameter among the declarations, we print out the type of the parameter and the parameter itself and mark that declared item to be a parameter. Finally, we go through the declarations and handle those items of the declaration which are not marked. However, because the grammar structure corresponding to declarations is divided into several classes, it is not easy to follow the algorithm as such.

We apply the solution presented in the previous example. Now we have to divide the C-compiling routine into three parts: the first part handles formal parameter, the second other variables and the third procedure definitions. Actually that part which handled procedure definitions remains unchanged, only the previous first part has to be divided into two new parts.

In order to implement the described algorithm we introduce a global list (global lists are described more detail in chapter 5). An object of class `Formal_parameters` puts each parameter into a global list. Then an object of class `Procedure_definition` goes through the global list of parameters, and at each item in the list performs the first C-compiling routine (which compiles parameters) of declaration objects. In the C-compiling routine, it checks if current declaration is in the current position of the parameter list. If it is, it writes appropriate C code to the output file and marks that item of the declaration to be a parameter, otherwise it does nothing. The second C-compiling routine object `Declaration` compiles only those items of declarations which are not marked as parameters. The described algorithm is not very effective, because every step of the parameter list requires going through the declarations. However, declared elements can be in any order, even parameters can be in different order than they are in the parameter list, and with this solution we manage to write parameters in correct order. In fact, there are usually only few items in the parameter list, and most of the necessary procedure calls during the traversal of the parameter list are calls of empty procedures.

4.2. Language structures having no counterparts in C

There are some language structures in PL/M which have no counterparts in C. If the translation from PL/M to C had been done by hand, it would be possible to convert each occurence of that kind of a structure in a manner that suits best that particular situation. When translating with an automatic tool we have to examine different practices to use these structures and find out which of them is the most general and covers best all the possible cases. When we have structures with no counterparts in the target language, the selection of the translation manner depends also whether the generality or precision of the translation is emphasized.

In this section we introduce two structures which have no counterparts in C: AT and BASED structures. These structures have been used in PL/M to locate several variables in the same places in memory. Both of them can be used in the implementation of variable-length records. BASED structure can also be seen as a pointer without type.

Example 4. AT structure

We distinguish two cases of using AT structure: a variable can be located at a component of an array variable or at an array variable or some other variable. First we consider the second case, in PL/M we can have the following declarations:

```
DECLARE
      LOCAL_BUFFER( 100 )                 BYTE,
      L3PARA_DATA1( 10 )
            STRUCTURE(  COUNTER( 3 )      BYTE,
                        THRESHOLD( 3 )    BYTE,
                        STATUS( 4 )       BYTE )
            AT ( @LOCAL_BUFFER ),
      L3PARA_DATA2( 10 )
            STRUCTURE(  YKM_DATA( 6 )     BYTE,
                        GEN_INFO( 2 )     WORD )
            AT ( @LOCAL_BUFFER );
L3PARA_DATA1( 2 ).COUNTER( 3 ) = 5;
```

The above declaration in PL/M means that both the array `L3PARA_DATA1` and the array `L3PARA_DATA2` are located in the same place as the array `LOCAL_BUFFER`. The same effect in C code can be achieved by using union type in the following way:

```
typedef struct
{
      unsigned char counter[ 3 ];
      unsigned char threshold[ 3 ];
      unsigned char status[ 4 ];
} generated_type_1;
typedef struct
{
      unsigned char ykm_data[ 6 ];
      unsigned shor gen_info[ 2 ];
} generated_type_2;
typedef union
{
      unsigned char local_buffer[ 100 ];
      generated_type_1 l3para_data1[ 10 ];
      generated_type_2 l3para_data2 [ 10 ];
} generated_type_3;
generated_type_3 generated_var_1;
generated_var_1.l3para_data1[ 2 ].counter[ 3 ] = 5;
```

The above code is slightly different from that proposed by Mikkonen [Mikk91]. The advantage of the above code is that it is more simple to generate from the corresponding PL/M code than Mikkonen's proposal.

The translation goes as follows. We just collect the variables located at the same place into a list (again we use a global list). At the end of the declaration we print out each item of that list as a field of a union. In addition, we have to know when we should omit the printing of a variable declaration, in this case this kind of variable is `local_buffer` in the top of the declaration. This can be done by examining beforehand the whole declaration and checking if a variable has been used as a location reference. Finally, the reference of the variable declared by AT attribute in PL/M program is different from its reference in C program, consequently we have to put a record reference before such a variable. (In the above example the record reference is `generated_var_1`.)

However, variables declared by AT attribute can be located also at a component of an array variable. In PL/M we can have the following declarations:

```
DECLARE
      PCB_BLOCK( 512 ) BYTE,
      TOT_ROW_COUNT     DWORD AT ( @PCB_BLOCK( 94 ) ),
      ROW_COUNT         DWORD AT ( @PCB_BLOCK( 98 ) ),
      BLOCK_COUNT       DWORD AT ( @PCB_BLOCK( 102 ) ),
      BYTES_LEFT_OVER   DWORD AT ( @PCB_BLOCK( 106 ) );
BLOCK_COUNT = 5;
```

Exactly the same effect in C program can be achieved by the following code:

```
unsigned char pcb_block[ 512 ];
#define TOT_ROW_COUNT   (* (unsigned int *)(pcb_block + 94));
#define ROW_COUNT       (* (unsigned int *)(pcb_block + 98));
#define BLOCK_COUNT     (* (unsigned int *)(pcb_block + 102));
#define BYTES_LEFT_OVER (* (unsigned int *)(pcb_block + 106));
BLOCK_COUNT = 5;
```

The advantage in this translation is that the usages of variables declared by AT attribute can be translated in the normal way. For example, when variable `BLOCK_COUNT` has been used in an assignment statement, it can be translated by normal assignment statement. The corresponding undefinitions have to be generated at the end of the scope. Another way to translate the above PL/M code is to use union type as we did in the first case:

```
typedef struct
{
      unsigned char dummy[ 94 ];
```

```
            unsigned int tot_row_count;
            unsigned int row_count;
            unsigned int block_count;
            unsigned int bytes_left_over;
      } generated_type_1;
      typedef union
      {
            unsigned char pcb_block[ 512 ];
            generated_type_1 generated_var_1;
      } generated_type_2;
      generated_type_2 generated_var_2;
      generated_var_2.generated_var_1.block_count = 5;
```

The above proposal is developed from Mikkonen's [Mikk91] translation. Some slight changes have been made in order to facilitate automatic translation. In this proposal the first field of the record is a dummy array, because the first part of the array `pcb_block` has not been used as a location. The variables declared with AT attribute are defined so that between their locations in the array remains no empty space. If there were any empty space, we would have to declare more dummy arrays between the locations. Consequently, the solution has the disadvantage that we should examine the size of each type of location. In our PL/M-to-C converter we have chosen the first approach if a variable is located at a component of an array.

Example 5. BASED structure

Another example of a structure which has no counterpart in C is BASED structure. It can be seen as a substitute of pointers without a type. In PL/M there is a pointer type, which can point to any type. Further declarations using BASED structure can define a type to the pointer variable. In PL/M we can have the following declaration:

```
      DECLARE
            PROG_STATUS_PTR POINTER,
            PROG_STATUS BASED PROG_STATUS_PTR BYTE;
      PROG_STATUS = SUCCESS;
```

The above declaration corresponds the following C declaration:

```
      unsigned char *prog_status_ptr;
      *prog_status_ptr = success;
```

In automatic conversion it is desirable that the same translation scheme suits in every cases. The above translation does not. If we want as general translation as possible we can use the following translation:

```
      void* prog_status_ptr;
      *(byte*) prog_status_ptr = success;
```

The above translation is logically the same as the corresponding PL/M code. We first declare a pointer with no type and then in each appearance of the pointer variable we change its type with casting. This translation suits to more complicated situations, too. For example, we can have the following declaration:

```
DECLARE
      LOCAL_BUFFER( 100 )                 BYTE,
      BUFFER_PTR                          POINTER,
      L3PARA_DATA1 BASED BUFFER_PTR ( 10 )
            STRUCTURE(  COUNTER( 3 )      BYTE,
                        THRESHOLD( 3 )    BYTE,
                        STATUS( 4 )       BYTE ),
      L3PARA_DATA2 BASED BUFFER_PTR ( 10 )
            STRUCTURE(  YKM_DATA( 6 )     BYTE,
                        GEN_INFO( 2 )     WORD );
BUFFER_PTR = @LOCAL_BUFFER;
L3PARA_DATA1( 2 ).COUNTER( 3 ) = 5;
```

It can be translated in an analogical way:

```
unsigned char local_buffer[ 100 ];
void* buffer_ptr;
typedef struct
{
      unsigned char counter[ 3 ];
      unsigned char threshold[ 3 ];
      unsigned char status[ 4 ];
} generated_type_1[ 10 ];
typedef struct
{
      unsigned char ykm_data[ 6 ];
      unsigned shor gen_info[ 2 ];
} generated_type_2[ 10 ];
buffer_ptr = local_buffer;
(*(generated_type_1*) buffer_ptr)[ 2 ].counter[ 3 ] = 5;
```

This kind of translation is very easy to make automatically, unlike Mikkonen's [Mikk91] corresponding translation. However, when selecting the translation approach we should take into account the desired type of the translator. If the translator need not care of how the generated C code looks like the later alternative can be chosen. If the intention is that the generated C code looks like hand written C code, we should handle different occurences of BASED structure in different manner or choose one best resembling usual C code and leave the other cases without handling at all.

In our converter we have chosen the alternative described in the last example. In future we will implement different alternatives of translation of the BASED structure and let the user to decide what manner is the best in each situation.

5. Problems caused by the implementation tool

The translator has been implemented using TaLE. When the conversion project began, TaLE was not tested very detailed yet. The PL/M-to-C converter was the first real software implemented using TaLE. This situation caused some problems to the project.

One of the principles of TaLE is to divide language structures into parts that are as independent as possible. They have only minimal connections to other language parts (nonterminals). A nonterminal can refer to its components, but not further; it cannot refer directly to the components of its own components. For example, a declaration structure is divided into several components, and even if those components in program text are rather close to each other, in a structural sense they can be very far away from each other. When we have situations in which a nonterminal has to know something about other nonterminals not close to it, we can solve the problem by two ways, either using transfer routines or a global object.

Transfer routines

Transfer routines act as follows. When transferring some information from one object to another indirect client of the first object, every object between those two objects must have a transfer routine. For example, in PL/M we are able to initialize variables by giving an initial value within parentheses. When initializing an array there are several initial values within parentheses. In C we have the corresponding rule. The only difference is that we use curly brackets merely in the case of an array. Assosiated to the above example PL/M has the following rules:

```
Variable_declaration –> Variable_name [Array_specifier]
                        Variable_type [Variable_attribute]
Variable_attribute > Public_attribute | External_attribute
Public_attribute –> [“PUBLIC”] [Locator] [Initialization]
External_attribute –> [“EXTERNAL”] [Initialization]
```

An object of the class `Initialization` should know if object `Variable_declaration` has an array specifier or not. According to the principles of TaLE those objects have no information about each other. The necessary information can be passed by using transfer routines. The object `Variable_declaration` must have in its C-compiling routine the call of the transfer routine `set_array`:

```
-- Variable_declaration

c_compile is
```

```
        do
              if not Variable_attribute_1.Void and
                 not Array_specifier_1.Void then
                    Variable_attribute_1.set_array;
              end;
              -- necessary statements
        end; -- c_compile
```

In TaLE/Eiffel the possible usage of optional structures can be tested by testing if the corresponging attribute is `Void`, which means that the object has not been created. In order to transfer that information object `Variable_attribute` has to have a transfer routine called `set_array`, which for example the class `External_attribute` can redefine by the following way:

```
  -- External_attribute

  set_array is
        do
              if not Initialization_1.Void then
                    Initialization_1.set_array;
              end;
        end; -- set_array
```

The features of `Initialization` are as follows:

```
  -- Initialization

  is_array: BOOLEAN;

  set_array is
        do
              is_array:=True;
        end; -- set_array

  c_compile2 is
        if is_array then
              output.putstring(“{ “);
        end;
        -- necessary statements
        if is_array then
              output.putstring(“ }”);
        end;
  end; -- c_compile2
```

In this case we could have passed the information by a parameter of the C-compiling routine. But in that case the C-compiling routines of different classes would have different parameters, which is not desirable.

Global object

The solution described is rather simple if the number of intervening objects is small (one or two). However, there are situations where that number is even seven or eight. So, the described solution is not always very practical. In such cases we can solve the problem with a global object. All the other objects know the state of that global object and they can also change its state. We can imagine the global object as some kind of symbol table which is used in generating the target code, not in analyzing the source code. (To analyze the source code TaLE has a symbol table mechanism of its own.) A global object can be used in passing information, the sender object sets options of the global object and the receiver object checks the information. Typically we can use this global object as a list store. One of these lists is the parameter list mentioned earlier.

Global object contains also other lists, one of which is the list of reserved words of C. We have the rule that variable names in C code should be the same as they were in PL/M. But because the reserved words are not same in the languages, a reserved word of C may have been used as a variable name in PL/M. Because of that those variable names should be changed. There are also other situations where the name of the variable must be changed. If we have in PL/M a simple variable, but we must change it to a record variable, the reference to the variable is different. In this kind of situations we put these variables to the list of changed names.

Each time we write an identifier to output file, we have to be aware if the identifier should be written in upper or lower case letters. It is desirable in C to use upper case letters if an identifier is defined with #define directive and otherwise lower case letters. Because of that we have to pick up all the identifiers defined by #define directive to a list and every time we are printing out an identifier we must check if it is in the list of upper case identifiers. The scope of the variables declared by #define directive is different from the scope of other variables. Consequently, we have to print out corresponding #undef directives at the end of the scope (procedure). For this purpose the global object has a procedure stack.

Here, we showed some examples of the features of the global object. Typically the global object contains different kinds of lists and routines which check if an item given as a parameter belongs to that particular list. With the global object it has been easy to handle the connections between the parsing objects. These objects are able to use the global object via its access class. The access class of the global object has only a once routine which returns the reference to the global object.

Problems caused by the implementation tool can be reduced to problems associated with information passing. When we consider a program as an abstract syntax tree, we note that there are two different situations in information passing. First, we can pass information between a parent nonterminal and a child nonterminal, in both directions. Second, we can pass information from a nonterminal to its coming sibling or its preceding sibling (or more generally between two nonterminals having same ancestor). In the second situation we pass the information through the same ancestor of the nonterminals. Thus, in this second situation we first pass the information from a descendant nonterminal to an ancestor, and then from the ancestor to another descendant nonterminal. In the second situation we apply two times the information passing of the first situation. If we reduce the information passing further, we only have the situation where information is passed between ancestor and descendant nonterminals in both directions. In every case, the decision which information passing way to choose, depends on the length of the path from the sender and receiver nonterminals.

The current version of TaLE does not provide special mechanisms for information passing, but possibly in future versions these problems will be taken into account. For transfer routines there might be a mechanism by which it would be possible to refer further than to the adjacent nodes in the syntax tree, for example to the next node (object) belonging to a certain class. For the global object there might be a mechanism which allows the user to specify a global object in a higher-level manner.

6. Conclusions

In this report we have described a converter from PL/M to C which has been implemented with TaLE. We have demonstrated how the described converter is different from the other existing converters. The differences is mainly due to TaLE. This project can also be seen as a testing of TaLE. The PL/M-to-C converter was the first real implementation by using TaLE.

During this work TaLE has turned out to be very dexterous tool. First of all the amount of the code needed for the translation is very small. Most of the operations have only a few lines. Although some algorithms have to be distributed to several classes still finding program errors was quite easy. When the translation was not expected, it was easy to define the nonterminal which caused the erroneous behaviour and find the corresponding C-compiling routine. Unfortunaly, the reuse feature of TaLE was not very easy to apply in this situation because of the PL/M grammar which is rather peculiar.

However, there are some shortcomings in TaLE from the PL/M-to-C converter's point of view. We used a global object and because of that we had to make a little adding to the Eiffel code generated by TaLE. The adding was just adding the access class of the global object to the inherit clause of each class which used the global object. The language implementor is able to do such addings, because TaLE is an open system. (TaLE can be even used as an Eiffel editor.) The shortcoming due to these addings is that each time the user makes changes to a class using the global object, (s)he must do the adding again, because TaLE generates the class after each changes. Because of that it would be desirable that there were some support for these global objects or for handling object lists in some other way.

TaLE has also other shortcomings due to the first version. TaLE and also Eiffel are rather slow. For example, even very little programs (ten lines or less) take about thirty seconds to translate. The next version of TaLE will be written in C++, and hopefully this improvement also makes the converter translate more quickly.

All in all, TaLE suits well to this kind of projects. The implementation of the converter using TaLE was probably easier than by using more conventional systems. The implemented PL/M-to-C converter will be further developed. One major improvement will be the adding of the user selections.

Acknowledgements
This work is supported by Ministry of Education and by the Academy of Finland through grant 1061120. I wish to thank Kai Koskimies and Esa Järnvall for the advice concerning TaLE and the translation itself.

References

[AGGH80] Albrecht, P. F., Garrison, P. E., Graham, S. L., Hyerle, R. H., Ip, P., Krieg-Bruckner, B., Source-to-source translation: Ada to Pascal and Pascal to Ada, Proceeding of the ACM-Sigplan Symposium on the Ada Programming Languages, Boston, Massachusetts, 1980, *Sigplan Notices* **15** (11) 1980 183 - 193.

[Frea80] Freak, R. A., A Fortran to Pascal translator, *Software - Practice and Experience* **11** 1981, 717 - 732.

[JäKo93a] Järnvall, E., Koskimies, K., Language implementation model in TaLE, Department of Computer Science, University of Tampere, Report A - 1993 - 1.

[JäKo93b] Järnvall, E., Koskimies, K., Computer-aided language implementation with TaLE, Department of Computer Science, University of Tampere, Report A - 1993 - 4.

[Mart86] Martin, D. G., Non-Ada to Ada conversion, *Ada Letters* **6** (1) 1986, 72 - 81.

[Meye88] Meyer, B., *Object-Oriented Software Construction*, Prentice - Hall, 1988.

[Mikk91] Mikkonen, A., The translation of DX 200 software from PL/M to C (in Finnish), Internal Report, Technical University of Helsinki, 1991.

[MoWa91] Moynihan, V. D., Wallis, P. J. L., The design and implementation of a high-level language converter, *Software - Practice and Experience* **21** (4) 1991, 391 - 400.

[PBBC88] Parsian, M., Basdell, B., Bhayat, Y., Caldwell, I., Garland, N., Jubanowsky, B., Robinette, J., Ada translation tools development: automatic translation of Fortran to Ada, *Ada Letters* **8** (6) 1988, 57 - 71.

[SlWa83] Slape, J. K., Wallis, P. J. L., Conversion of Fortran to Ada using an intermediate tree representation, *The Computer Journal* **26** (4) 1983, 344 - 353.