Cobol Data Flow Restructuring

Gerbrand Stap

August 11, 2005

Master Software Engineering,
University of Amsterdam

Supervisors:
Steven Klusener [1]
Niels Veerman [1]
Mark van den Brand [2]

Performed at: Free University of Amsterdam, May–August 2005

[1]Free University of Amsterdam
[2]University of Amsterdam

Contents

Abstract

In 1997 the world-wide amount of COBOL code was estimated at 180 GLOC (1 GLOC = 1,000,000,000 lines of code), which was growing at a rate of 5 GLOC a year. If this growth did not change in the past eight years, it would mean that COBOL accounts for 220 GLOC world wide(!) [2]. This in contrast to the amount of newly trained COBOL developers which – according to [7] – is decreasing.

To cope with the vastly increasing amount of code that has to be maintained by the shrinking group of COBOL developers and maintainers, some measures have to be taken that enhance the maintainability of the COBOL code. One possible approach to this is the reduction of global variables. The transformations presented here can determine parameters and local variables in existing COBOL code.

1 Introduction

1.1 Rationale

In the first standard of COBOL – released in 1960 – the only variables that could be declared were global variables. The reason for this is that there is one specific division reserved for declarations, the `DATA DIVISION`. The big problem with one large block of variable declarations is that later on a variable can be used everywhere in the program. So it is not clear which variable belongs to which part of the code. Also it is not clear where a variable will or might be changed. The solution to this problem is a block structured program with methods, procedures and/or functions, which is very common in modern languages. These blocks can contain their own variable declarations and their scope is limited to that block.

Another problem of programs that have no block structure is the lack of parameters. In order to 'simulate' parameters in COBOL, some assignments (`MOVE`s) are typically done before and after a call of a section (`PERFORM`). For example:

```
MOVE OWN-VAL-1 TO ADD-VAL-1
MOVE OWN-VAL-2 TO ADD-VAL-2
PERFORM ADD-UP
MOVE ADD-RESULT TO OWN-RESULT.
```

It is clear that this is more lengthy and more variables are required than with single procedure call with two input and one output parameter. Removing unneeded `MOVE` statements and declarations does not belong within the scope of this research, but is a logical follow-up.

Block structured programs were introduced in COBOL with the COBOL-85 standard (ISO/IEC 1989:1985). The concept that gave the programmer the opportunity to make use of local variables and parameters was called *'nested programs'*. Unfortunately, twenty years later, this technique is still not used on a large scale. Some recent compilers also support OO COBOL, which is part of the COBOL-2002 standard (ISO/IEC 1989:2002) and is also used in the Microsoft .Net framework. In OO COBOL, classes can be defined which contain methods that have their own local variables and parameters.

To attack the problem of the global variables in COBOL, automatic transformations will be used to transform 'normal' COBOL programs into COBOL programs that make use of local variables and parameters. The reason why automatic transformations are used is the consistency of the resulting code, the possibility to improve and apply the transformations rapidly and the fact that knowledge about what is transformed is stored in the transformations (and not in difference between the original and the resulting code).

So the aim of this research becomes the increase in maintainability of COBOL programs by improving its data flow with the use of automatic transformations. The result should be a block structured program that makes use of parameters and local variables, and with the same functionality as the original program. The results will be an overview of the parameters and local variables that are resolved in order to create such a block structured program and of course the resulting code.

1.2 Nested programs vs. OO COBOL

As mentioned, there are two ways in COBOL to make use of local variables and input and output parameters.

Nested program are (as the term implies) programs within another program. These nested programs look the same as 'normal' programs: they have their own `DATA` and `PROCEDURE DIVISION`s and can be called with the `CALL` statement, which is also used to pass input and output parameters. The main difference with a normal program is that a nested program is only accessible from the containing program.

The structure of an OO COBOL program is quite different from a normal program, because it is composed of classes. Instead of sections and paragraphs, the `PROCEDURE DIVISION` of a COBOL class contains methods. Methods also have their own `DATA DIVISION` and can accept input and output parameters. From that point of view, methods are like nested programs. But when declared `PUBLIC`, methods are accessible by other programs. Methods are called using the `INVOKE` statement, which like a `CALL` statement can accept input and output parameters. When comparing nested programs to methods, (containing) programs should be comparable with classes. This comparison is not correct though, because a class can not be called, only its methods can be `INVOKE`d.

It is the intention to take existing code, transform it to improve its data flow, and use it in its original environment. That is why it is desirable that the transformed code is compatible with the original code. If a program would be transformed to a class, the `CALL`s to that program should be changed to `INVOKE`s to its methods and it would be necessary that the (existing) compiler supports OO COBOL. Because of these incompatibilities, it is easier to stick to a more conventional solution and therefore the transformation to nested programs is discussed here. A transformation to OO COBOL could be done later with the use of the resolved parameters and local variables, if needed.

1.3 Related work

A great part of the algorithm that extracts parameters and local variables is derived from [5]. An algorithm that extracts parameters from RPG code for the generation of wrappers is presented in that paper. For this research, this algorithm is adapted for COBOL code and some features are added to extract the local variables.

In [4] the problem of implicit structures is recognized. Such structures are for instance: enumerated types, procedures and local variables. In [4] some leads are given to make these implicit structures explicit. Also a step by step plan is given to 'remodularize' COBOL code (create nested programs). This plan is, however, not exhaustive and is meant to be carried out by hand. The transformations presented here do the same work in an automated manner.

The main benefits and bottlenecks of nested programs are discussed in [1]. Performance issues are also addressed and the conclusion is made that calling a nested program is faster when less than five parameters are used. And frequently called nested programs are most likely faster than frequently performed sections / paragraphs. But the difference in performance is probably hardly measurable. Note that improving performance is not a goal for this research.

A research similar to the research presented here is described in [12]. In this research, global variables are identified in Ada code. When such a global variable is used within a procedure, the variable will be added as a parameter of this procedure. This is done for the same reason as the transformations applied to the COBOL code, namely to enhance the maintainability of the code. The difference is that the variable in the Ada code will remain global and the identification of parameters is only done to indicate that a variable is used in a specific procedure. The analysis to determine the 'global parameters' is based on only one procedure at a time, so it is not checked whether a value set to a variable will reach a use in another procedure. This may result in unneeded input and/or output parameters.

More research, mainly concerning automated transformations on COBOL code, at the Free University in Amsterdam can be found in [6]. A part of this research is presented in [11], where COBOL/CICS legacy systems are transformed to make them better maintainable. The main operations done on COBOL code are the removal of `GO TO` statements and the isolation of paragraphs that can be used as subroutines. This 'componentization' can be seen as a first step towards transforming COBOL programs into block structured programs.

A research that focuses on COBOL and is based on [11], is presented in [13]. The transformations from [11] are improved in [13] to make them more stable and more transparent and they are combined in an automated algorithm that can be applied on a large amount of COBOL programs. Also more transformation rules were implemented so that they could be applied to more COBOL programs.

Automatic transformations are also applied on real-world COBOL code at the Free University. A good example of this can be found in [8]. These transformations concerned a data expansion of a product code that needed to be changed from two to three digits.

The papers [11] and [13] present transformations to enhance the control flow of a COBOL program (e.g. remove `GO TO`s, etc.). This can be seen as a prerequisite to enhance the data flow, which can be done using the transformations presented here.

1.4 Technical details

The transformations are specified using the ASF+SDF Meta-Environment. The ASF+SDF Meta-Environment is an interactive development environment for the automatic generation of interactive systems for manipulating programs, specifications, or other texts written in a formal language. The generation process is controlled by a definition of the target language, which typically includes such features as syntax, pretty printing, type checking and execution of programs in the target language [3].

In this case an already defined grammar, specified in SDF, of COMPAQ COBOL was used. This grammar was obtained earlier using a IBM VS II COBOL grammar [10], which was adapted using the Grammar Deployment Kit (GDK)[9]. Beside it, some grammars were specified in SDF for intermediate data structures specific to this research. Rewrite rules to transform the COBOL programs and to determine the local variables and parameters, were specified in ASF.

Comments will disappear when using a compiled version of the ASF transformations. Removing comments does not increase maintainability, which is one of the main motivations for this research. According to the website [3], it will be possible to match layout and comments in future versions of the ASF+SDF Meta-Environment (from version 1.6, which is currently under development).

All examples concerning grammars / syntax and rewrite rules are given in the ASF+SDF formalism.

A code base of 1.2 MLOC real-world COMPAQ COBOL code is available for testing purposes and will be addressed simply with the term ‘code base’ from here on.

Source code examples were tested using IBM ILE COBOL/400 on an IBM AS/400.

Transformations were made and run using a desktop PC with an AMD Athlon 64 3200+ processor (running at 2200 MHz) and 512 MB memory.

1.5 Organization of thesis

The thesis will first continue with ‘Algorithm and its implementation’ (section 2) in which – at first – some assumptions are discussed that were made before the algorithm was implemented. After that the algorithm and how it was implemented in the transformations are discussed in four steps (2.3 through 2.6). A number of subjects that were encountered and caused the need to adapt the implementation are discussed in ‘Points of interest’ (2.7).

The next and last section is ‘Results’ (section 3). In this chapter some statistics are given that were collected during the appliance of the transformations on a real life system. This is followed by a few topics that could not be solved for the time being, or with other words ‘Future work’ (3.2). Finally, a conclusion (3.3) completes this thesis.

2 Algorithm and its implementation

2.1 Assumptions

In order to successfully apply the transformations, two assumptions are made. These assumptions are explained in this section.

Program consists of at least two sections

The `PROCEDURE DIVISION` (the statement part) of a Cobol program must consist of sections, or paragraphs optionally followed by sections. Sections, on their turn, can also contain paragraphs. So a program does not need to have sections, but the assumption states that they do. This is because each section will be transformed to a nested program. The reason for this is that all programs in the code base are completely divided into several sections. Each section performs a logical function like opening a file, writing data to it or displaying results. So there already is a division into functionality and these sections are therefore logical candidates to be transformed nested programs. These sections also contain paragraphs, but it is not logical to transform these paragraphs to nested programs. Some sections, for example, are divided in two paragraphs, where the first paragraph is used to initialize variables and the second part processes these values. Transforming such paragraphs to nested programs would result in two nested programs that exchange all variables that they use. Transforming a program that contains only one section is quite useless, because the first section will not be transformed to a nested program and thus the program would not be modified (see also section 2.6).

A Cobol program that is not divided into sections or just has one section would require some preprocessing in which possible sections are identified. Or the transformation could be adapted so that paragraphs are transformed instead of sections. But this is not within the scope of the work presented here.

Paragraphs in other sections are not accessed directly

In order to execute the statements in a section and let the program return to the executing statement (just as one would expect with a function or method), this section needs to be `PERFORM`ed. An interesting detail of Cobol is that the `PERFORM` statement can also be used on paragraphs, which can exist inside a section. `GO TO` statements can also jump to a section or paragraph, with the difference that the flow of control will not automatically return to the `GO TO` statement.

Execution of a (nested) program is done by using the `CALL` statement, which can transfer parameters. `PERFORM`s on sections will be translated to these `CALL`s on nested programs and `PERFORM`s on paragraphs will be left unchanged. After the transformation, there might still be paragraphs inside a nested program, but these paragraphs are no longer reachable from other nested programs. This is the reason why `PERFORM`s and `GO TO`s on paragraphs inside another section (and later: another nested programs) are not allowed. A `PERFORM` or `GO TO` on a paragraph within the same section will not cause any problems and will be moved (without change) to the new nested program.

2.2 Overview

The transformation of normal Cobol programs to programs that contain nested programs, is actually a series of transformations. The transformations that must be applied to the code in order to acquire the final result are (section numbers are included):

2.3 Code normalization

2.4 Data flow extraction

- Variable declarations
- Data and control flow

2.5 Data flow analysis

- Build data flow tables
- Determine parameters and local variables

2.6 Transformation to nested programs

To determine the parameters and local variables, there must be knowledge of the existing variables. Therefore all declarations of the program that will undergo the transformation are gathered first. This data is needed in all remaining transformations. The data and control flow can now be extracted from the statements of the program. The data gathered here are the 'uses' (the value of a variable is used) and 'sets' (the value of a variable is set) of the variables and some simplified control flow information.

After these steps the actual algorithm that determines the parameters and local variables, using the data and control flow information, is applied. This algorithm is derived from the algorithm presented in [5]. The two main steps in this algorithm are to build up eight tables that contain specific information about the data flow, and determining the parameters and local variables.

All transformation will be discussed in detail in the following corresponding sections.

2.3 Code normalization

Some code normalizations are applied to the code which make the transformation less complex and therefore better understandable. Some examples of these normalizations are the removal of optional (meaningless) keywords and the addition of `ELSE` branches to `IF` statements. In order to force these code normalizations, some of the transformations from [13] are applied before analysis are done. The only reason for the appliance of these normalizations is the simplification of the analysis that determine the parameters and local variables. They will not be visible in the resulting code.

Nameless variables and `FILLER`s

A normalization that was not addressed in [13] is the removal of nameless variables and `FILLER`s. The problem with nameless variables and `FILLER`s is that they are often used in `REDEFINES`, for example:

```
01 YEAR PIC 9999.

01 REDEFINES YEAR.
   03 CENTURY PIC 99.
   03 FILLER  PIC 99.
```

REDEFINES are used to define different classifications for the same data. So the YEAR variable and the nameless variable make use of the same piece of memory. Changing the YEAR changes the CENTURY and vice versa. Therefore this notation is comparable to typecasting. In this code example it is impossible to fully qualify the variable CENTURY and to store the redefine of YEAR (because it can not be named). That is why a transformation was made that renamed nameless variables to names that were temporarily added to the syntax (NONAME#x and FILLER#x), to avoid name collisions. So the example declaration would become the following during transformation:

```
01 YEAR PIC 9999.

01 NONAME#1 REDEFINES YEAR.
   03 CENTURY PIC 99.
   03 FILLER#2  PIC 99.
```

During transformation the redefined variable can be referred to as NONAME#1 REDEFINES YEAR and it is possible to fully qualify unnamed variables, like CENTURY IN NONAME#1. Of course, these names will not be visible in the resulting code, because they do not comply to the COBOL syntax.

Copy books

Copy books must be expanded before the code can be analyzed. The code base contained mostly copy books that inserted a declaration of some frequently used data structure. This data is of course crucial for the output of the algorithm. That is why copy books were expanded with use of a pre- and post-processing script, written in Perl.

2.4 Data flow extraction

Variable declarations

A very typical characteristic of COBOL are the record structures that can be declared and used as variables. These records mostly represent a logical set of data that reflects real world data. To keep these logical sets of data logical to a programmer, the records have to be kept together. Take for example a postal code record like this:

```
01 POSTALCODE.
   03 DIGITS     PIC 9999.
   03 CHARACTERS PIC XX.
```

When a subroutine only uses the DIGITS of the POSTALCODE, only these DIGITS *could* be passed as an input parameter. But it is unclear what a parameter named DIGITS represents. That is why the algorithm must only produce parameters and local variables that are top-level (01) variables. So in this case the entire POSTALCODE will be passed, even if only the DIGITS are required.

Data flow restructuring is all about the variables, so the way in which the variable declarations are stored during transformation is crucial. The record structure of variables is a very important factor in the representation, because it must be clear when a variable is set. When, for example, a postal code is declared as in the example in this section and used as follows:

```
SOME SECTION.
  ...
  MOVE "1081HV" TO POSTALCODE
  DISPLAY DIGITS.
  ...
```

It becomes clear that there has to be knowledge about all declared variables. The representation of the variables must make a distinction between levels of records. The distinction between levels is necessary because it has to be clear when the value of a variable is set. It is set when its own value is set, the value of a super-level is set, or all sub-level variables are set. The variable DIGITS in the example is therefore set before use and it will not be necessary to, for example, pass DIGITS as input parameter for section SOME.

During analysis COBOL variables are represented by using a built-in list (of the Meta Environment). This list is used to store the fully qualified name of the variable by putting the name of the bottom level in the list as the first element. This way it is easy to determine whether for example a variable has a value set to it by using list matching. The ASF+SDF Meta Environment also has a built-in set, which is used to store all variable declarations. The syntax in SDF of a COBOL variable and the set of all variables is:

```
"[" { Type "," }* "]"             -> List[[ Type ]]
"{" { Type "," }* "}"             -> Set [[ Type ]]

List [[ Lex-cobword ]]            -> Variable
Set  [[ List[[ Lex-cobword ]] ]] -> Variable-set
```

Where the Lex-cobword is a sort from the COBOL grammar that is used (amongst other things) for a name of a variable. Note that a variable is represented by a list of Lex-cobwords, i.e. its fully qualified name. After the declarations of the earlier mentioned POSTALCODE is gathered, it will thus be represented like this:

```
{[POSTALCODE], [DIGITS, POSTALCODE], [CHARACTERS, POSTALCODE]}
```

A number sections in the DATA DIVISION can contain variables, but only variables from the WORKING-STORAGE SECTION are gathered. Variables from the FILE SECTION are ignored, because file descriptors can not be passed between (nested) program. Variables from the LINKAGE SECTION (in which parameters are declared) are also ignored, because the interface of the program must not change. More details on this can be found in section 2.7.

Redefines are also gathered and stored in a built-in table of the Meta Environment. The general syntax of this table and the syntax of the specific table used, are:

```
List [[ < Key, Value > ]]      -> Table[[ Key, Value ]]

Table[[ Variable, Variable ]]  -> Redefine-table
```

In this table, the first variable (the key) is the variable that redefines the second variable (the value).

Here is another example of a postal code (with another name to avoid any confusions), but now the digits are accessible using a redefine. Note that the original code is given and the code after the transformation mentioned in 2.3, because otherwise it would not be possible to store the redefines.

Declarations	**After renaming**
`01 POSTAL PIC 9999XX.`	`01 POSTAL PIC 9999XX.`
`01 REDEFINES POSTAL.`	`01 NONAME#1 REDEFINES POSTAL.`
`   03 DIGITS PIC 9(4).`	`   03 DIGITS PIC 9(4).`
`   03 REDEFINES DIGITS.`	`   03 NONAME#2 REDEFINES DIGITS.`
`      05 FIRST2 PIC 99.`	`      05 FIRST2 PIC 99.`
`      05 LAST2  PIC 99.`	`      05 LAST2  PIC 99.`
`   03 PIC XX.`	`   03 NONAME#3 PIC XX.`

The resulting Redefine-table would be:

Redefine-table
`[ < [NONAME#1], [POSTAL] >,`
`  < [NONAME#2, NONAME#1], [DIGITS, NONAME#1] >`
`]`

This table is used to check, for every use or set of a variable (see next paragraph), whether it has a redefine. If this is the case, not this variable will be used in the remaining part of the algorithm, but the variable it redefines. So making use of the variable `NONAME#1` is interpreted as using `POSTAL`, and using `NONAME#2` is interpreted as using `DIGITS`. When the program is actually transformed to a program with nested programs, all level `01` redefines have to be kept together with the variables they redefine. Redefines on lower levels are automatically kept together because they will belong to the same super levels.

Data and control flow

All necessary data regarding the data and control flow will be gathered in this stage. Only two possible facts about a variable used in the source code have to be retrieved: the set or use of a variable. The representation of the control flow is also limited to the minimum; a `PERFORM` of another section is stored (perf) and it is possible to indicate that the control flow path splits up (fork). The syntax of the needed data is as follows:

```
"set"  Variable             -> Dataflow-info
```

```
"use"  Variable             -> Dataflow-info

"perf" Label-name           -> Dataflow-info
"fork" "("
    Dataflow-info-list ":"
    Dataflow-info-list
")"                         -> Dataflow-info

List [[ Dataflow-info ]]    -> Dataflow-info-list
```

Where `Label-name` is a sort from the COBOL grammar that is used for a name of a section.

The explanation of this notation will be done using some examples:

`ADD` statement

Statements	`Data-flow-info-list`

`ADD A, B TO C, D`	`[ use[A], use[B], use[C],`
	`  set[C], use[D], set[D]`
	`]`

The `ADD` statement given here performs two calculations, namely: `C` = `C` + `A` + `B` and `D` = `D` + `A` + `B`. The most important aspect here is to get the uses and sets in the right order. The values of `C`, `A` and `B` will be used before `C` is set and in addition `D` is used to set `D`. It is not a problem that '`use[A], use[B]`' is not repeated before '`use[D]`', because the results that matters (e.g. the value of `A` is used before the value of `D` is set) will remain unchanged. Such a conclusion makes a difference when a statement like '`ADD A TO A`' (= `A` + `A`) is used, because `A` is now used before it has been set. When `A` is not set in previous statements, it will probably become an input parameter of the section where where this statement is used.

`IF` and `MOVE` statement

Statements	`Data-flow-info-list`

`IF C = 0`
`  MOVE B TO A`
`ELSE`
`  MOVE C TO A`
`END-IF`
	`[ use[C], fork(`
	`    [ use[B], set[A] ] :`
	`    [ use[C], set[A] ]`
	`  )`
	`]`

The `IF` statement is probably the most obvious example to show the use of the '`fork`'. After checking the condition (`C = 0`), there are two possible paths to execute, the if-branch and the else-branch. Both paths consist of a `MOVE` statement, which uses the value of its first argument to set the value of the second.

Record structures

Statements	`Data-flow-info-list`

`MOVE "1081HV"`
`  TO POSTALCODE`
`DISPLAY DIGITS.`
	`[ set[POSTALCODE],`
	`  use[DIGITS, POSTALCODE]`
	`]`

This example uses the declaration of a postal code from section 2.4 and illustrates that a variable from a lower level will be represented with its fully qualified name.

`PERFORM` **statement**

Statements	`Data-flow-info-list`
`PERFORM VARYING A`	`[ use[B], set[A], use[C],`
`        FROM B`	`  fork(`
`        BY D`	`    [ perf SEC, use[A],`
`        UNTIL C`	`      use[D], set[A],` *`use[C],`*
`    PERFORM SEC`	*`      fork(`*
`END-PERFORM`	*`        [ perf SEC, use[A],`*
	`          use[D], set[A]`
	`        ] :`
	`        [ ]`
	`      )`
	`    ] :`
	`    [ ]`
	`  )`
	`]`

This example contains a few interesting things. First of all it contains two types of `PERFORM` statements, the in-line (1st) and the out-of-line (2nd) `PERFORM`. The in-line `PERFORM` produces a counter `A` that starts at `B` and is each loop increased by `D` as long as condition `C` is true *before* the statements are executed. The out-of-line `PERFORM` calls another section or paragraph and is transformed to '`perf SEC`'. All other data flow information is generated by the in-line `PERFORM`.

The representation of the control flow is chosen to simplify the processing of the data and control flow at a later stage. Therefore the in-line `PERFORM` must be represented with only the `fork` mechanism. The condition (`C`) can be false before the first loop, therefore a `fork` is inserted after the use of the condition which has one empty path. The other path contains the data and control flow information of the statements nested in the in-line `PERFORM` (i.e. `perf SEC`), followed by the increase of the counter (i.e. `use[A], use[D], set[A]`). In order to match the data and control flow information with the semantics of a loop, the part from the use of the condition to the end of the `PERFORM` statement is copied (the part in italics). Now all possible sequences of uses and sets are represented in the data and control flow information. Even when the loop is always executed more than twice, it is not necessary to copy the part in italics more often, because it would not add any new sequences, which is the only thing that matters for the representation of the control flow.

For a complete program the data and control flow information is stored in a table that uses the name of the section (the `Label-name`) as key and the `Dataflow-info-list` as value. Thus, the syntax of the table is:

```
Table[[ Label-name, Dataflow-info-list ]] -> Dataflow-info-table
```

By using this `Table` it becomes easy to look up the `Dataflow-info-list` of a specific section. This comes in handy when, later on in the algorithm, this data must be looked up for a performed section. Here is an example of how a piece of code is transformed to a `Dataflow-info-table`:

Example with more sections

Sections	Data-flow-info-table
SOME SECTION.	[< SOME,
MOVE 'HELLO WORLD!' TO S	[set[S], perf DISP]
PERFORM DISP.	>,
	< DISP,
DISP SECTION.	[use[S]]
DISPLAY S.	>
	]

2.5 Data flow analysis

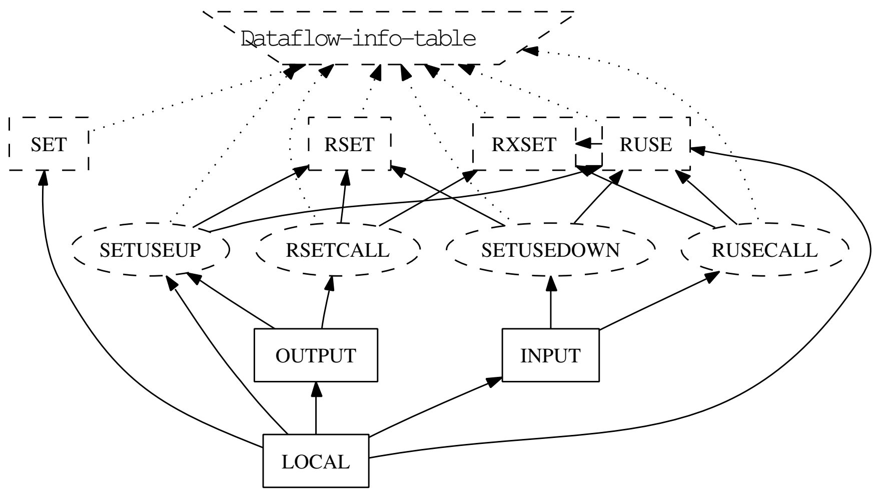
The algorithm used to resolve the input and output parameters from existing code, is the algorithm described in [5]. This algorithm is used to resolve the interface of RPG program subroutines to produce wrapper code. This problem is comparable to the problem in this case, with the difference that the knowledge about the resolved interface is used to restructure existing code instead of producing wrappers.

The algorithm presented in [5] is based on the control flow graph of the program, and therefore called flow-sensitive. This approach is more precise (and more intensive) than a flow-insensitive algorithm like the one presented in [12]. This is feasible because the transformations on the code are needed to enhance the maintainability. When, for example, too many parameters are added to a subroutine, maintainability decreases because it still is not clear what that subroutines exactly needs to do its work properly.

A restriction of the algorithm is that it can not handle programs with recursive subroutines. The good news is that not all COBOL compilers can handle recursive PERFORMs of sections or paragraphs mainly because recursive sections or paragraphs are not a part of the official COBOL standard. The only way in COBOL to define a recursive process is to make an entire program recursive (by adding the `RECURSIVE` phrase) and let that program `CALL` itself. But since the scope of this research is not on interaction between programs but between sections, this restriction of the algorithm causes no problems.

Build data flow tables

The first step in the algorithm is to build up a number of tables that contain information about the data flow. This information is gathered using the extracted information about the control and data flow from the `Data-flow-info-table` (see section 2.4) and will eventually be used to determine the parameters and local variables. In this first step the eight tables RUSE, RXSET, RSET, SET, SETUSEDOWN, SETUSEUP, RUSECALL and RSETCALL will be built. These tables will then be used to build the tables INPUT, OUTPUT and LOCAL, which hold the input parameters, output parameters and local variables respectively. The dependencies between these tables (and the `Dataflow-info-table`) is shown in figure 1.



Nodes that have the same shape are tables with the same syntax.
Dashed nodes are intermediate results.

Figure 1: Dependencies of data flow tables

The tables RUSE, RXSET, RSET, SET, INPUT, OUTPUT and LOCAL have the following syntax:

```
Table[[ Label-name, Variable-set ]]
```

This means that for each section (with name `Label-name`), a set of variables is stored that have a specific characteristic. How this variable set is determined, will be explained later on. The remaining tables (SETUSEDOWN, SETUSEUP, RUSECALL and RSETCALL) have the syntax:

```
Table[[ < Label-name, Label-name >, Variable-set ]]
```

These tables contain a variable set for each pair of section names (the tuple `< Label-name, Label-name >`), where the first section performs the second.

Note that the tables are different from the the way they are defined in [5]. The choice of the syntax for these tables relies mostly on the used tool, the ASF+SDF Meta Environment.

Each table will be explained in the following paragraphs.

RXSET contains the variables that are (indirectly) set on all paths in a section. Its contents vary during the process of building up the tables. The following example will be used to explain the tables RXSET, RUSE, RSET and SET:

Code example for RXSET, RUSE, RSET and SET

Declarations

```
01 X PIC X.
01 Y PIC X.
01 Z PIC X.

01 CONDITION PIC 9.
```

Code fragment

```
A SECTION.
  MOVE 'x' TO X
  IF CONDITION = 0
    MOVE 'y' TO Y
    * checkpoint
  END-IF
  DISPLAY X
  PERFORM B.

B SECTION.
  DISPLAY X
  DISPLAY Y
  MOVE 'z' TO Z.
```

Intermediate RXSET table at comment '`* checkpoint`'

```
[ < A, { [X], [Y] } > ]
```

At the 'checkpoint' `X` and `Y` are always set in section `A`.

RXSET table at end of code fragment

```
[ < A, { [X], [Z] } >,
  < B, { [Z] } >
]
```

At the end of the code fragment control flow is of course outside the `IF` statement, so it is possible that `Y` is not set and therefore deleted from the RXSET table. On the other hand, the variable `Z` is set (and added to RXSET) in section `B`, so the `PERFORM` of `B` in section `A` also causes the addition of `Z` to the RXSET table for section `A`.

RUSE contains for each section a set of variables that are (indirectly) used without being set on every path. To check whether a variable is set, the RXSET table is consulted (see previous paragraph). The RUSE table for the example will thus be:

RUSE table

```
[ < A, { [CONDITION], [Y] } >,
  < B, { [X], [Y] } >
]
```

In section `B` the variables `X` and `Y` are not set before they are `DISPLAY`ed. So these two variables are added to the RUSE table for `B`. Note that it is not taken into consideration that section `A` is executed first. In section `A` the value of `CONDITION` is never set and thus is added to the table. `X` is present in the RXSET table at the first statement '`DISPLAY X`', so `X` must not be added to the RUSE table for section `A`. The same applies to the indirect use of `X` in section `B`. `Y` is indirectly used outside the `IF` statement, because `B` is `PERFORM`ed by `A`. So `Y` is not set on all paths leading to the '`DISPLAY Y`' statement. This is the reason for the addition of `Y` for section `A`.

RSET contains for each section all variables that might have been (indirectly) set. When looking at the example, the difference with the RXSET table is that `Y` is never removed:

RSET table
`[ < A, { [X], [Y], [Z] } >,`
`  < B, { [Z] } >`
`]`

SET (among other tables) is later used to determine local variables. As the name indicates, it is the same as the RSET table without the indirectly set variables. So when using the same example, the result will be:

SET table
`[ < A, { [X], [Y] } >,`
`  < B, { [Z] } >`
`]`

The SET table is the only table that is not presented in [5], because only the problem of determining input and output parameters for wrappers is covered and no local variables are determined.

Before explaining the rest of the tables, another form of the explained tables is given here. This form contains all variables from the original table plus all sub-levels of these variables. The notation of this form is the name of the original table with an asterisk (*). The definition is (taking the RSET table as example):

$$RSET^*(S) = \{x | y \in RSET(S), x \in \texttt{Variable-set}; x \subseteq y\}$$

In this definition '$x \subseteq y$' means that x is a variable that is part of the record structure y, or the same variable. This form is only used here to keep the definitions of other tables more compact and clear.

Two other notation used in the definitions indicate the state (and thus contents) of a table before ($\rightarrow$`perf` S) and after (`perf` $S \rightarrow$) the perform (`perf`) of section S.

SETUSEDOWN contains for each pair of a performing (S_1) and a performed section (S_2) the set of variables that might be set in the performing section before the actual `PERFORM` statement and then used in the performed section. So the definition becomes:

$$SETUSEDOWN(S_1, S_2) = RSET^*(S_1, \rightarrow\texttt{perf } S_2) \cap RUSE(S_2)$$

These variables will become input parameters for the performed section. For an example, see figure 2.

SETUSEUP can be seen as the opposite of the SETUSEDOWN table and contains for each pair of a performing (S_1) and a performed section (S_2) the set of variables that might have been set in the performed section and after the `PERFORM` statement used in the performing section. So the definition becomes:

$SETUSEUP(S_1, S_2) = RUSE(S_1,\texttt{perf}\ S_2 \rightarrow) \cap RSET^*(S_2)$

These variables will become output parameters for the performed section. For an example, see figure 2.

RUSECALL contains for each pair of a performing (S_1) and a performed (S_2) section the set of variables that is used by the performed section but not set by the performing section before the actual `PERFORM` statement. The definition of this table is:

$RUSECALL(S_1, S_2) = RUSE(S_2) \setminus RXSET^*(S_1, \rightarrow\texttt{perf}\ S_2)$

These variables might become pass through variables by being an input parameter for both sections (S_1 and S_2). For an example, see figure 2.

RSETCALL contains for each pair of a performing and a performed section the set of variables that is set in the performed section, but does not get set in the performing section after the `PERFORM` statement. The definition of this table is:

$RSETCALL(S_1, S_2) = RSET(S_2) \setminus RXSET^*(S_1, \rightarrow\texttt{perf}\ S_2)$

These variables might become pass through variables by being an output parameter for both sections (S_1 and S_2). For an example, see figure 2.

Example

When the code from figure 2 would be transformed to a program with nested programs, it should contain two variables (`X1` and `X2`) that are passed immediately from the section where they are set to to the section where they are used. Two other variables (`Y1` and `Y2`) have to be passed through by section `B` in order to get same output as in the original form of the program. This variable passing is indicated by the 'arrows' on the right side of the asterisks.

To give a complete picture, all earlier explained tables are also given for this example:

RUSE table
`[ < B, { [X1], [Y1] } >,`
`  < C, { [Y1] } >`
`]`

RXSET and RSET table
`[ < A, { [X1], [Y1], [X2], [Y2] } >,`
`  < B, { [X2], [Y2] } >,`
`  < C, { [X2], [Y2] } >`
`]`

Note that RXSET and RSET are the same because of the lack of statements that influence the control flow (e.g. an `IF` statement that would result in a `fork`).

SETUSEDOWN table
`[ < <A, B>, { [X1], [Y1] } > ]`

```
ID DIVISION.
PROGRAM-ID. EXAMPLE.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 X1 PIC X(2).
01 X2 PIC X(2).
01 Y1 PIC X(2).
01 Y2 PIC X(2).
01 S  PIC X(6) VALUE IS 'value '.

PROCEDURE DIVISION.
A SECTION.
    MOVE 'X1' TO X1    *  ->X1
    MOVE 'Y1' TO Y1    *     |  ->Y1
    PERFORM B          *     |     |
    DISPLAY S Y2.      *     |     |  <-Y2
    STOP RUN.          *     |     |     |
                       *     |     |     |
B SECTION.             *    \/    \/    /\
    DISPLAY S X1       *  <-X1     |     |
    PERFORM C          *           |     |
    DISPLAY S X2.      *  <-X2     |     |
                       *     |     |     |
C SECTION.             *    /\    \/    /\
    MOVE 'X2' TO X2    *  ->X2     |     |
    DISPLAY S Y1       *        <-Y1     |
    MOVE 'Y2' TO Y2.   *              ->Y2
```

Figure 2: Code example for SETUSEDOWN, SETUSEUP, RUSECALL and RSETCALL

These variables will become input parameters (from `A` to `B`). They are added because they exist in the RUSE table of `B` and in the RSET table of `A` at the point where `B` is performed. The variable `Y1` is not directly used in `B`, but has to be passed through to `C`. This will be determined later using the values in the RUSECALL table.

SETUSEUP table

`[ < <A, B>, { [Y2] } >,`
`  < <B, C>, { [X2] } >`
`]`

These variables will become output parameters (from `B` to `C`). The entries are added because they are set in the performed section and then used in the performing section. In the case of the entry '`<<B, C>, [X2]>`', this is quite obvious. The other entry (`<<A, B>, [Y2]>`) seems a bit odd, because `Y2` is not directly set in `B`. Instead, `Y2` is indirectly set in `B` through the `PERFORM` of `C` and thus also needs to be an output parameter of `C`. This will be determined later using the values in the RSETCALL table.

RUSECALL table

```
[ < <B, C>, { [Y1] } > ]
```

The fact that `B` has to pass through `Y1` from `A` to `C`, can be derived from this entry. As it happens, the only entry in the table is also a useful one. But the table may contain unneeded entries, like the RSETCALL table.

RSETCALL table

```
[ < <A, B>, { [X2], [Y2] } >,
  < <B, C>, { [X2], [Y2] } >
]
```

The only value needed from this table is the value `Y2` for the key `<B, C>`. The values in this table are compared to those from the SETUSEUP table according to the definition given earlier. Because section `A` is never called, the first entry is unneeded. Section `B`, however, is called by `A` and `A` expects the output parameter `Y2` from `B`. But `B` does not set `Y2`, so `B` has to pass through this variable, which can be concluded from the second entry.

Determine parameters and local variables

The resulting input (INPUT table) and output parameters (OUTPUT table) can now be determined; they are already indicated by the arrows in the comments of figure 2. In the example only 01 level variables are used, but it is possible that the tables presented so far contain variables of other levels. In 2.4 it is mentioned that only 01 level variables can become a parameter or local variable. **So from this point, only the 01 level of variables in the tables are needed**. This means that all entries of variables in the previous presented tables must be replaced by their 01 level. Otherwise the following definitions are not correct.

In the definitions of INPUT and OUTPUT, SETUSEDOWN and SETUSEUP are used with a single argument. In this form the table contains only values for the performed sections (S_2). So when a performed section has more than one entries, these entries are ‘unioned’.

$$SETUSEDOWN(S_2) = \bigcup_{S_1 \in \text{fanin}(S_2)} SETUSEDOWN(S_1, S_2)$$

Note that the ‘fanin’ function on S_2 results in a set of all sections that perform S_2. The definitions of the INPUT and OUTPUT tables are:

$$INPUT(S) = SETUSEDOWN(S) \cup \left(\bigcup_{S_1 \in \text{fanin}(S)} SETUSEDOWN(S_1) \cap RUSECALL(S_1, S) \right)$$

$$OUTPUT(S) = SETUSEUP(S) \cup \left(\bigcup_{S_1 \in \text{fanin}(S)} SETUSEUP(S_1) \cap RSETCALL(S_1, S) \right)$$

In these definitions a clear division between the direct and pass through parameters is visible. The direct parameters are the sets SETUSEDOWN(S) and SETUSEUP(S). The rest of the definitions produce the pass through variables.

The definition of the LOCAL table is:

$$LOCAL(S) = SET(S) \cup \left(\bigcup_{S_1 \in \text{fanout}(S)} SETUSEUP(S, S1) \right) \setminus RUSE(S) \setminus INPUT(S) \setminus OUTPUT(S)$$

Note that the 'fonout' function is used here, which produces a set of all section called by its argument. The candidate local variables are those variables that are set before use in particular section [4]. That is why the variables from SET are taken and those from RUSE are removed. A variable must also become local when a section S receives an output parameter P from a section it performs and S does not have the output parameter P. These are typically the variables from the SETUSEUP table for the performing section. A variable can never be both parameter and local variable for the same section, so variables from INPUT and OUTPUT are also removed.

According to the definition, the INPUT table for the example from figure 2 has to be:

INPUT table

```
[ < B, { [X1], [Y1] } >,
  < C, { [Y1] } >
]
```

The first entry contains in this case all 'direct' input parameters, taken from the SETUSEDOWN table. The only indirect input parameter in the example is `Y1`. This parameter is found when the entries from SETUSEDOWN and RUSECALL) are compared. The called section from the SETUSEDOWN (which is `B`) is equal to the calling section from RUSECALL. All similar variables from both entries (only `Y1` in this case) are added to the INPUT table for the called section from RUSECALL, which is `C`. These entries are in line with the input parameters indicated by the arrows in the example code.

According to the definition, the OUTPUT table for the example from figure 2 has to be:

OUTPUT table

```
[ < B, { [Y2] } >,
  < C, { [X2], [Y2] } >
]
```

For the OUTPUT table, direct parameters are taken from the SETUSEUP table. Indirect output parameters are determined in the same way as the input parameters, but instead the SETUSEUP and RSETCALL tables are used. A difference is that the RSETCALL table contains unused values (RUSECALL does not) which do not intersect with the values from the values from SETUSEUP.

According to the definition, the LOCAL table for the example from figure 2 has to be:

LOCAL table

```
[ < A, { [X1], [Y1], [Y2] } >,
  < B, { [X2] } >
]
```

2.6 Transformation to nested programs

The transformation to a program with nested programs can now be done with the data gathered and stored in the INPUT, OUTPUT and LOCAL table. The actual transformation has not yet been implemented, but some tests are done to determine whether a transformed program, is functionally equivalent to the original program. Transforming the program from figure 2, should result in a program similar to the program in figure 3:

The output of both the original program and the resulting program with nested programs is:

```
value X1
value Y1
value X2
value Y2
```

The most striking differences with the original program are, of course, the two sections that have become a nested program. The first section remained a section, because a COBOL program can not start with a nested program. An alternative for this is to transform all sections and generate a first section that `CALL`s the original first section. This would result in a little less global variables, because the original first section will also receive its own parameters and local variables. The two sections that were transformed, have the `COMMON` phrase added to their `PROGRAM-ID` paragraph. This enables the nested programs to be called by all nested programs within the same outer program, instead of only the program that directly contains the nested program. It might be that the `COMMON` phrase is not necessary for every nested program (like `B` in this case), but in the original program, all sections were reachable from all other sections. So adding the `COMMON` phrase to all nested programs resembles the original program the most.

There is one variables left that is not mentioned in the INPUT nor the OUTPUT nor the LOCAL table, namely `S`. When this is the case, such a variable should remain global. This can be done by adding the `GLOBAL` clause to its declaration, which makes a variable in the outer program accessible to all nested programs. The conclusion that can be made here is that variables that have no `GLOBAL` clause are local variables of the first section (`A` in this case).

In each nested program, the variables in the `DATA-DIVISION` are directly related to those mentioned in the resulting INPUT, OUTPUT and LOCAL tables. The local variables can be found in the `WORKING-STORAGE SECTION` and the input and output parameter are located in the `LINKAGE SECTION`. The parameters can also be found in the `USING` phrase of the `PROCEDURE DIVISION` header. The names in the `USING` phrase determine the order in which the parameters must be given in a `CALL` statement to that nested program and the `LINKAGE SECTION` defines the types of these parameters. In this example the input parameters are passed `BY CONTENT` and the output parameters are passed `BY REFERENCE`. It must be possible for a nested program to return more than one variable. IBM VS II COBOL has a `CALL` statement that has an optional `RETURNING` phrase, but this can only handle one output parameter. COMPAQ COBOL does not even contain this phrase, but has instead has a `GIVING` phrase that can only return an integer value. So there is no other possibility to return more than one variable than to pass the needed parameters `BY REFERENCE`.

```
ID DIVISION.
PROGRAM-ID. EXAMPLE.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 X1 PIC X(2).
01 Y1 PIC X(2).
01 Y2 PIC X(2).
01 S  PIC X(6) VALUE IS 'value ' IS GLOBAL.

PROCEDURE DIVISION.
A SECTION.
    MOVE 'X1' TO X1
    MOVE 'Y1' TO Y1
    CALL 'B' USING BY CONTENT X1 Y1
                   BY REFERENCE Y2
    DISPLAY S Y2.
    STOP RUN.

ID DIVISION.
PROGRAM-ID. B IS COMMON.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X2 PIC X(2).
LINKAGE SECTION.
01 X1 PIC X(2).
01 Y1 PIC X(2).
01 Y2 PIC X(2).
PROCEDURE DIVISION USING X1 Y1 Y2.
B SECTION.
    DISPLAY S X1
    CALL 'C' USING BY CONTENT Y1
                   BY REFERENCE X2 Y2
    DISPLAY S X2.
END PROGRAM B.

ID DIVISION.
PROGRAM-ID. C IS COMMON.
DATA DIVISION.
LINKAGE SECTION.
01 Y1 PIC X(2).
01 X2 PIC X(2).
01 Y2 PIC X(2).
PROCEDURE DIVISION USING Y1 X2 Y2.
C SECTION.
    MOVE 'X2' TO X2
    DISPLAY S Y1
    MOVE 'Y2' TO Y2.
END PROGRAM C.

END PROGRAM EXAMPLE.
```

Figure 3: Example in figure 2 transformed to a program with nested programs.

In order to execute the nested programs, the PERFORM statements must be transformed to CALL statements. A syntactical difference with the PERFORM statement is that the name of the called nested program must be placed within quotes (because the name can also be stored in a variable). The parameters that must be added to the CALL statement can easily be looked up in the INPUT and OUTPUT tables.

2.7 Points of interest

Existing parameters must remain unchanged after the transformation, otherwise the interface of the program would be compromised and the new program would be incompatible with the original. So the question is whether it is possible that the algorithm produces a result which would change the interface of the program. In theory this is possible, but only when the value of the parameter is set before use in all sections where the parameter is mentioned. Because a parameter is treated as a normal variable, it would, in that case, become a local variable in those sections. This would mean that the passed value for that parameter is never used in the program. The most plausible explanation for this would be that it is assumed that the parameter is always passed BY REFERENCE and therefore is an output parameter (just like the output parameters for the nested programs).

So the conclusion is that variables from the existing LINKAGE SECTION must never be transformed to local variables of nested programs. In the case that the algorithm does produce a result that states that a parameter is a local variable, it must instead receive a GLOBAL clause to its declaration. Existing parameters can however become parameters between nested programs, just like normal variables (which excludes that they will be local variables).

Initial values of variables, given in the declaration of those variables, are not taken into account during the algorithm. But this does not cause any difficulties, see for example the variable S in the example program from figure 2 and the resulting code in figure 3

The initial value can be seen as a sort of input parameter, so the same reasoning applies as for the existing parameters. With of course the difference that it can not be an output parameter of the program. Therefore it is not a problem when this variable would become local, but it means that its initial value is never used. This could be the case when, for example, all values are (by convention) set to a default value, like SPACES.

Files are different from variables because they can have a status like ‘opened’, ‘read-only’ and ‘closed’. This is not a problem when one section makes use of a particular file. But when different sections perform different operations on a file, like opening, writing and closing, files become a problem. Distributing operations on files is a common practice in the available code base and therefore a solution to this problem had to be found.

Passing the file descriptors (FDs) as parameters between the different nested programs is not an option, because this is not allowed [1]. So the best way to overcome the problem is to leave the file description, along with its record description, global. This means effectively that declarations of files and file records are not gathered in the algorithm. With adding the GLOBAL clause to the file description, the file will be accessible from all nested programs. The statements that do operations on the file can therefore be left unchanged.

Declaratives are the error-handling sections in COBOL. DECLARATIVES are often triggered when a file error occurs. Files will remain global, so most DECLARATIVES do not need to be examined to figure out if they might be moved to a nested program. This also avoids code duplication and unnecessary work. DECLARATIVES can also be made global by adding the GLOBAL clause. Variables used in those declaratives are also removed from the INPUT, OUTPUT and LOCAL table to ensure that they will will receive the GLOBAL clause and are thus reachable from both the nested programs and the declaratives.

Indexes can be declared along with multi-dimensional records, also called tables (comparable to an array). These indexes enhance the performance when using tables. But indexes do not maintain their value when passed as a parameter in a CALL statement. Therefore all indexes are also gathered and treated as if they were normal variables. So when an index is recognized as an input or output parameter, it must be separately passed and its value must be copied to the index of the passed table.

Database manipulation in COMPAQ COBOL can be done using the FETCH statements which occurred in some programs in the code base. This statement can also be used to retrieve a record that is on a higher level in hierarchical structure of the database, as in:

```
FETCH OWNER WITHIN A-SET.
```

Where OWNER is a reserved keyword that represents the higher level in hierarchical structure of the database. The problem is that the information about the structure of the database can not be derived from the code. So it is not known which variables are set or used when this statement is executed. The structure can be retrieved using the data description of the database, but this was too complex when taking in account the available time for this project.

False positives can occur when using the presented algorithm. Take for instance the following code example (a fragment of code from the code base actually did something like this):

```
...
IF X
    MOVE "DATA" TO Y
END-IF
IF X
    DISPLAY Y
END-IF
...
```

When displaying the contents of Y, the algorithm assumes that the value of Y does not necessarily have to be set. This is because the statement that sets this value (`MOVE`) is inside of a conditional statement (`IF`), and the statement that uses this value (`DISPLAY`) is inside another branch of another conditional statement. Even though the conditions for both the setting and using statements are the same, `Y` might now be identified as an input parameter, because the values of conditions are not analyzed. Whether `Y` will be an input parameter can not be concluded from the example, because there has to be a statement in another section that sets the value of `Y` and this value must reach the statements of the code example. The functionality of the program will not be compromised when `Y` does become an input parameter. But an unneeded parameter could cause confusion and therefore maintainability would decrease.

PERFORM THRU statements are used to execute a series of paragraphs and / or sections. Especially in combination with `GO TO` statements, these `PERFORM THRU`s can be very complex to understand. After an examination of the code base, the conclusion was that there was no direct need to address this problem, because there existed only two `PERFORM THRU`s. These `PERFORM THRU`s were also on paragraphs within the same section and thus could be ignored, because they would be placed within a new nested program and semantics would be left unchanged.

3 Results

3.1 Applying the transformations

To test the transformations and to see if the transformations produce a usable result, they were applied to a system from the code base that contained 84 programs suitable for transformation. Some programs in this system were skipped because they contained only one section and were too small to divide into more sections. A program with one section is not interesting because this section is not transformed to a nested program. To give an idea of the size of the test set, some characteristics are given:

- 84 programs
- 82,656 LOC (70,102 LOC non-comment and non-empty)
- 1,676 level 01 variables
- 1,296 sections
- 2,349 performs

The time consumed by the transformation varied greatly for each program and could take from 1 second to 4 hours and 40 minutes. This was not in direct relation with the lines of code of the program. A remarkable example is a certain program of 3,144 LOC took 16.65 seconds to be transformed, while another program of 2,177 LOC took 316.59 seconds (a factor 19 longer). The programs had an equivalent number of section (33 and 31 respectively) and calls to other sections (83 and 71 respectively). And while the `Dataflow-info-table` of the first program was almost twice the size of the second program, it still was much easier to retrieve the parameters and local variables of the first program. The reason for this is probably the complexity of the `Dataflow-info-table`, which is caused by factors like the number and depth of nested `fork`s.

The following table summarizes the results of the transformation:

type	amount	duplicates (%)	unique (%)	amount per section
input	2,125	1,633 (77%)	492 (23%)	1.64
output	1,769	1,306 (74%)	463 (26%)	1.36
local	880	460 (52%)	420 (48%)	0.68
total	4,774	4,018 (84%)	756 (16%)	3.68

A remarkable number from the table is the number of duplicates in the 'total' row (note that this is not the sum of the other duplicates, because one variable can be both parameter and local variable within the same program). According to this number each variable that becomes a parameter of local variable must be declared more than five times. This means that in the final form of the program 4,018 declarations will be duplicates of an already existing declaration. Code duplication is not desired when enhancing the maintainability of source code, but unfortunately in this case it will be necessary.

The number of unique variables that can be used as parameter or local variable seems a bit low, because it is only 16% of the total number of parameters and local variables. But when this number is compared to the original number of variables (1,676), it is actually quite high, because *on average 45% of all global variables in the original program can be used as a parameter or local variable!*

These numbers also show that an average section has an almost 'traditional' number of parameters; one or two input parameters and one, sometimes two, output parameters. The number of local variables is a bit disappointing, only two third of the sections has one local variable.

3.2 Future work

`GO TO`s

An observant reader might have noticed that the notorious `GO TO` statement was not mentioned in the section about control flow extraction (section 2.4). Many `GO TO` statements can be eliminated using the transformations from [13], but there is no guarantee that this will remove all `GO TO`s. `GO TO`s can be copied from a section to a nested program without compromising the control flow, provided that these `GO TO`s do not jump out of the section. This because paragraphs in other sections will no longer be reachable when these sections are transformed to nested programs. The problem with `GO TO`s within a section is that they can influence the control flow such that a section would require other parameters and local variables.

Analyzing a control flow that includes `GO TO`s is much more complex than analyzing a control flow represented as in section 2.4. A possible solution for this problem is to, at first, include the `GO TO`s and then expand these `GO TO`s in order to eliminate them from the control flow representation.

Two scenarios are presented here that are most common: a forward and a backward `GO TO` from a conditional statement. The forward `GO TO` is mostly used to skip some statements that must not be executed when some condition is met. In languages that do not support `GO TO`s, these skipped statements will most likely appear in an 'else' branch of an 'if' statement. The backwards `GO TO` is mostly used to create some loop, comparable to a 'for' or 'while' loop.

The expansion of a forward `GO TO` is visible in figures 4 and 5 and the expansion of a backward `GO TO`s is visible in the figures 6 and 7.

Removing unneeded `MOVE` statements and declarations

This is a next step to enhance the maintainability of COBOL programs, as mentioned in the introduction (section 1.1). In order to 'simulate' parameters in COBOL, some assignments (`MOVE`s) have to be done before and after a call of a section (`PERFORM`). So calling a section that adds the two variables `ADD-VAL-1` and `ADD-VAL-2` and stores the result in `ADD-RESULT`, could look like this:

```
MOVE OWN-VAL-1 TO ADD-VAL-1
MOVE OWN-VAL-2 TO ADD-VAL-2
PERFORM ADD-UP
MOVE ADD-RESULT TO OWN-RESULT.
```

Whereas calling a (sub/nested) program, using parameters looks like this:

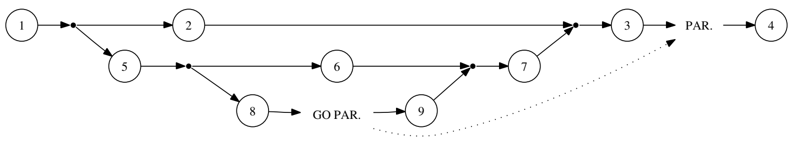


Figure 4: Expanding a forward GO TO in the control flow, original

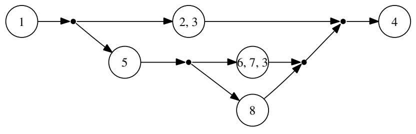


Figure 5: Expanding a forward GO TO in the control flow, result

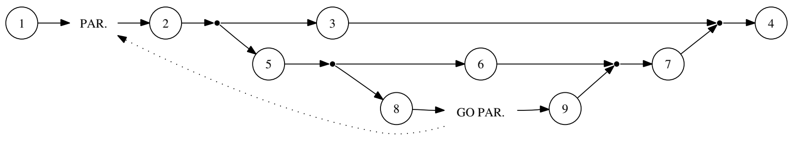


Figure 6: Expanding a backward GO TO in the control flow, original

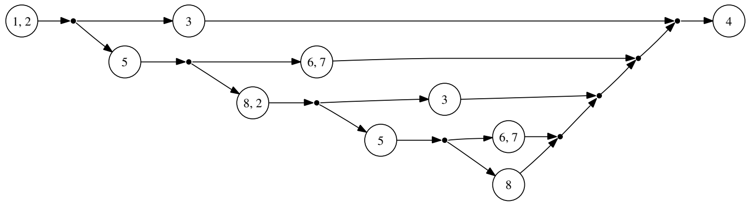
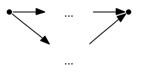


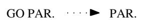
Figure 7: Expanding a backward GO TO in the control flow, result

Legend

Dataflow-info-list



fork



A GO TO with corresponding paragraph

```
CALL "ADD-UP" USING  OWN-VAL-1 OWN-VAL-2
              GIVING OWN-RESULT.
```

This form is better maintainable, because there is no misunderstanding about what values the called subroutine uses. This in contrast to the `PERFORM ADD-UP` statement, which gives no clarity about the variables used in the subroutine. The second form also differs in number of statements and variable declarations. It does not contain the three `MOVE` statements and the three variables `OWN-...` do not have to be declared.

Removing these `MOVE` statements and declarations is a great profit to maintainability, because it is often difficult to gain a clear view of the functionality of COBOL code because of the many `MOVE` statements and declarations. As an indication: the code from the code base exists of 831,168 LOC (1.2 MLOC minus comments and empty lines) which contains 93,945 (11%) declarations and 140,138 (17%) `MOVE` statements.

Tables and pointers

These are variables that can be dynamically addressed, which is in contrast to the static analysis done on the code. Tables are very often used in COBOL, but fortunately a quick scan of the code shows that most uses of tables are quite decent implemented; when a number of records are set in a section and this section `PERFORM`s a section that uses records from the same table, the used records are almost always the same or a subset of the records that are set. In other cases the resulting parameters and local variables could be incomplete, because the record number (or index) is ignored when the used variable is determined.

So in the following example:

```
WORKING STORAGE SECTION.
01 SOMETABLE.
   03 SOMERECORD PIC X(20) OCCURS 10 INDEXED BY IDX.
...
A SECTION.
  PERFORM VARYING IDX FROM 1 BY 1 UNTIL IDX > 10
    MOVE "SOMEVALUE" TO RECORDS(IDX)
  END-PERFORM.
  PERFORM B.

B SECTION.
  PERFORM VARYING IDX FROM 1 BY 1 UNTIL IDX > 5
    DISPLAY SOMETABLE(IDX)
  END-PERFORM.
```

The current algorithm only 'knows' that there is a variable `SOMERECORD` (and not `SOMERECORD(1) ... SOMERECORD(10)`). But since this variable is set in `A` and this value reaches the use in `B`, `SOMERECORD` will become an input parameter for `B`. The records used in `B` are a subset of those set in `A`, so there will be no changes in the functionality in this case. But it can not be guaranteed that in other situations, all operations on tables will remain functionally unchanged.

Pointers are rarely used in the code base and most of this small collection of pointers are initialized immediately when they are declared. These pointers are treated as redefines, which they in fact are as long as the pointer is not changed in the program.

An improved algorithm should fully analyze these types of variables, probably by interpreting the code to determine all possible values of pointers and record numbers. In some situations this might be possible (e.g. `PERFORM` statements that always process a complete table), but a generic approach to this problem would be complex to implement.

3.3 Conclusion

In this thesis a series of transformations was presented that make it possible to transform plain COBOL code into block structured COBOL code. In order to determine the needed parameters, the algorithm from [5] was adapted to COBOL and features were added to determine local variables. Most difficulties concerning the adaption of the algorithm to COBOL were solved, resulting in fully automated transformations that was applied to over 80KLOC of COBOL code. This resulted in some promising numbers. The most interesting fact was that almost half of the variables in the programs can be used as parameter or local variable. On average, for each block of code one or two input parameters and one, sometimes two, output parameters were found.

With the data resulting from the transformations presented here, the following step towards the transformation to block structured COBOL programs can be made.

References

[1] T. Scott Ankrum. Cobol nested programs. Object-Z Systems Inc
http://objectz.com/columnists/tscott/04302001.htm.

[2] Edmund C. Arranga and Frank P. Coyle. Cobol: Perception and reality. *Computer*, 30(3):126–128, 1997.

[3] M. van den Brand et al. The Asf+Sdf Meta-Environment.
http://www.cwi.nl/projects/MetaEnv/.

[4] Tony Cahill. Remodularization of COBOL programs through Scope and DataFlow Analysis. University of Limerick
http://www.csis.ul.ie/RsrchPubs/Remodcob.htm.

[5] Aniello Cimitile, Ugo de Carlini, and Andrea de Lucia. Incremental Migration Strategies: Data Flow Analysis for Wrapping. In *Working Conference on Reverse Engineering (WCRE)*, pages 59–68, 1998.

[6] Cobol Research at the Free University in Amsterdam.
http://www.cs.vu.nl/Cobol/.

[7] Bill C. Hardgrave and E. Reed Doke. Cobol in an Object-Oriented World: A Learning Perspective. *IEEE Software*, 17(2):26–29, March/April 2000.

[8] S. Klusener, R. Lämmel, and C. Verhoef. Architectural Modifications to Deployed Software. *Science of Computer Programming*, 54:143–211, 2005.

[9] J. Kort, R. Lämmel, and C. Verhoef. The grammar deployment kit. *Electronic Notes in Theoretical Computer Science*, 65(3), 2002.

[10] R. Lämmel and C. Verhoef. VS Cobol II Grammar Version 1.0.4, April 2003.
http://www.cs.vu.nl/grammars/vs-cobol-ii/.

[11] M. P. A. Sellink, H. M. Sneed, and C. Verhoef. Restructuring of COBOL/CICS legacy systems. *Science of Computer Programming*, 45(2–3):193–243, 2002.

[12] Ricky E. Sward and A. T. Chamillard. Re-engineering global variables in ada. In *SIGAda '04: Proceedings of the 2004 annual ACM SIGAda international conference on Ada*, pages 29–34, New York, NY, USA, 2004. ACM Press.

[13] N. Veerman. Revitalizing modifiability of legacy assets. *Software Maintenance and Evolution: Research and Practice, Special issue on CSMR 2003*, 16(4–5):219–254, 2004.