Testability Analysis Applied to Embedded Data-flow Software

Thanh Binh Nguyen, Michel Delaunay, Chantal Robach
LCIS-ESISAR, BP 54, 50, rue B. de Laffemas, 26902 Valence, France
{Binh.Nguyen-Thanh, Michel.Delaunay, Chantal.Robach}@esisar.inpg.fr

Abstract

Testability is an important quality factor of software, particularly embedded data-flow software such as avionics software. A lack of testability of such software can badly affect test costs and software dependability. Testability analysis can be used to identify parts of software which are difficult for testing.

In this paper, we propose the use of the Static Single Assignment (SSA) form to transform source code generated from data-flow designs into a data-flow representation, and then we describe some algorithms to automatically translate the SSA form into a testability model. Thus, some metrics can be applied to the testability model in order to locate the software parts which induce a weakness of the testability.

Keywords: *Testability Analysis, Software Measurement, Data-flow Software.*

1. Introduction

Software development is more and more complex, hence quality assurance needs more effort, in which testing is a crucial task. During this task, faults must be revealed. However, testing increases the reliability of the software, but it never can ensure that there are no faults in the software.

Particularly, testing of embedded data-flow software, such as avionics software, is very expensive in terms of cost and complexity. This is why testability metrics have been defined these last years in order to help in appraising the ease (or difficulty) for testing software.

First, some software complexity measures were considered as a substitute of testability, such as McCabe's metric [8] or Nejmeh's metric [9]. Then, some specific research works tackled the testability concept. Freedman [3] introduced the testability measures for software components by defining observability and controllability notions, these testability measures are only applied to functional specifications by examining input and output domains. Voas and Miller [14] also proposed a testability metric based on the inputs and outputs domains of a software component. Then, they proposed the PIE (Propagation, Infection and Execution) technique to analyze software testability in [15]; this technique can be only applied to the source code. Le Traon and Robach [6, 7] proposed testability measures particularly for dataflow designs, which are graphically represented by a diagram of operators (or components). In the communication software area, Petrenko and *al.* [11] investigated testability of communication software which is modeled by a composition of finite state machines, then Karoui and *al.* [5] proposed a testability metric for communication software modeled by relations. Jungmayr [4] presented testability measurement in the context of static dependencies within object-oriented software. Each one of these methods was investigated in order to analyze testability within a specific application area.

In this paper, we focus on testability measurement of source code generated from data-flow designs. These measures can be used 1) to identify parts of a design which contribute to a lack of testability, 2) to compare the design testability and the generated code testability in order to evaluate the code generation process, and 3) to improve the software reliability. A testability analysis is essential for embedded data-flow software such as avionics software; as these software parts are often designed with a data-flow approach, this paper is essentially concerned with data-flow software.

In a previous work, Le Traon and Robach [6, 7] proposed a testability model, which is implemented in the SATAN tool (System's Automatic Testability ANalysis), to analyze testability of data-flow software. The proposed testability model is a bipartite oriented graph, it was shown to be applicable to model data-flow designs. However, it cannot be applied to analyze the source code generated from the designs or component code used in the designs: because the source code is often generated or described in imperative languages. So, we recently proposed the use of the Static Single Assignment (SSA) form to apply this model for testability analysis of component code [10]. Code generated from a data-flow design is indeed an integration of code of the components used in the design. So, it is possible to use the SSA form as an intermediate representation of source code to analyze the code testability. Then, we propose some algorithms to automatically translate the SSA form into testability model, which is used to compute the testability measures for the source code.

Section 2 briefly presents the testability model and the testability measures which are based on it. Section 3 outlines the SSA form and presents the algorithms to automatically translate the SSA form into the testability model. Section 4 describes how to apply the SATAN tool to the case study and presents the results. Finally, Section 5 gives our conclusion.

2. Testability analysis

In this section, we present the testability model and the testability measures for data-flow designs [6, 7], which were implemented in the SATAN tool.

In this approach, the *testability* of a software is based on the controllability and the observability of the software components. *Controllability* is defined as the ease to bring the inputs of the software to the inputs of a component. *Observability* is defined as the ease to propagate the outputs of a component to the final outputs of the software.

The testability model is a directed graph representing the information transfer within the software. Then, the testability measures are computed on this model, by appraising the information quantity through the associated graph.

We illustrate the construction of the testability model on the example of a data-flow design (provided by THALES Avionics) in Figure 1.

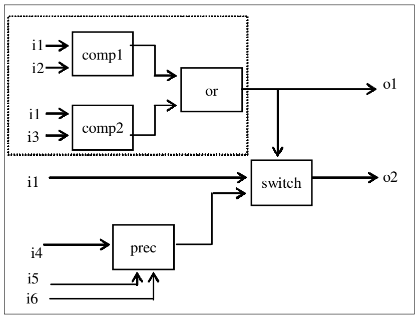

Figure 1. A data-flow design

In this diagram, $o1$ and $o2$ are a couple of outputs, $o1$ is a boolean output serving to verify whether $o2$ is a valid output; $i1$, $i2$, $i3$, $i4$, $i5$ and $i6$ are the inputs; *comp1* and *comp2* are the comparison components; *or* is a logical component; *switch* is a selection component; and *prec* is a memorisation component.

2.1. Testability model

The principle of modeling the information transfers through software consists of representing the control and dataflow aspects on a same graph that is called the *Information Transfer Graph* (ITG). It is a bipartite directed graph. This model is defined by places, transitions and edges. The places are:

- the modules, which are operators or components;
- the inputs, which are inputs for the software;
- the outputs, which are observable results of the software.

The inputs and outputs are terminal nodes. The transitions represent the mode of information transfer between the places. Three basic modes of information transfer are considered for modeling a data-flow design (Figure 2):

- junction mode: when the destination place needs information from both source places;
- attribution mode: when the destination place needs information from either one of several source places;
- selection mode: when the same information goes from the source place to several destination places.

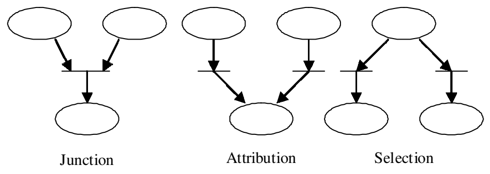

Figure 2. Information transfer modes

The edges connect the places and the transitions.

In a graphic representation, modules are represented by circles; inputs and outputs, by semicircle; transitions, by bars.

The ITG of the above example is given in Figure 3.

The Information Transfer Graph contains all control points, as in a conventional control flow graph and also all relations between definitions and uses of variables, as in a conventional data flow graph. Thus, this model is particularly suitable for dataflow specification languages.

2.2. Flows

The ITG is used to identify information paths, which are called *flows*, throughout the software. A *flow* is an information path from some inputs to one output. It contains a set

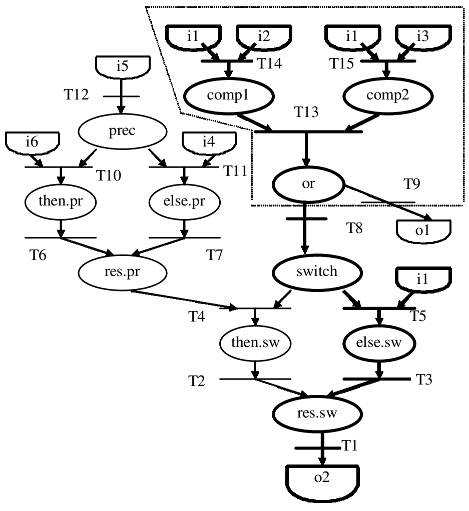

Figure 3. Information Transfer Graph

of places, transitions, and edges. Thus, a flow can be considered as a subgraph; it is an elementary function that can be independently exercised from the remainder of the software, since it computes the output variables from the inputs variables.

Note that in an ITG, a module node is similar with an "or" node, while a transition node is similar with an "and" node. So, to exercise a transition node, all predecessor nodes must be exercised (junction mode), while a module node is exercised when one of the predecessor nodes is exercised (attribution mode or selection mode). This principle allows the computation of flows from some inputs to an output in ITG.

In the above graph, four flows are identified. For sake of simplicity, each flow is characterized by the set of modules it contains and the output it computes $F_i = \{$*modules* $\mid$ *output*$\}$:
$F_1 = \{$*or, comp1, comp2* $\mid o1\}$
$F_2 = \{$*res.sw, else.sw, sw, or, comp1, comp2* $\mid o2\}$
$F_3 = \{$*res.sw, then.sw, res.pr, sw, then.pr, or, prec, comp1, comp2* $\mid o2\}$
$F_4 = \{$*res.sw, then.sw, res.pr, sw, else.pr, or, prec, comp1, comp2* $\mid o2\}$
For instance, flow F_2 is drawn in bold in Figure 3.

2.3. Test strategies

Once the set of flows is identified, it is used to determine the set of test objectives according to a test strategy. A test strategy is an ordered set of flows which must be exercised through the software. A test strategy corresponds to a test data selection criterion. The selection criterion is to cover every module in the model at least once by executing the selected flows.

Two test strategies are used in SATAN: *Multiple-Clue* and *Start-Small*. The *Multiple-Clue* strategy is based on choosing a subset of flows that satisfy coverage of all the modules: all flows chosen are exercised, possible information of fault is collected, and diagnostic is analyzed on this information. This strategy is effective in the case of simple faults (only one module is defective). The *Start-Small* strategy is based on a gradual coverage of the modules by choosing flows with an increasing complexity in terms of the number of covered modules, and a new flow is tested only if faults detected in previous flow are corrected; a minimum subset of flows is chosen so that all the modules are covered. This strategy is effective in the case of multiple faults (several modules are defective). So, applying test strategies allows the number of flows to be reduced in terms of cost while insuring that all modules in the ITG are covered.

2.4. Testability measures

Testability is based on the controllability and the observability of a module for each flow of the software. The controllability measure estimates the information quantity available on the inputs of a module from the inputs of the software through the considered flow. Respectively, the observability measures estimate the information quantity available on the outputs of the software from the outputs of a module (Figure 4). To compute the information loss through the flows, we need to introduce the module capacity concept. A module capacity is the information quantity that is available on the module outputs from its inputs: this expresses the information loss through the module.

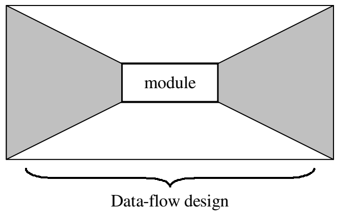

Figure 4. Controllability and Observability of a module

Let's consider:
I_F the variable representing the inputs of flow F;
O_F the variable representing the outputs of flow F;
I_M the variable representing the inputs of module M;
O_M the variable representing the outputs of module M.
The *controllability measure* of module M in a flow F is given by:

$$CO_F(M) = \frac{T(I_F; I_M)}{C(I_M)}$$

where T(I_F;I_M) is the maximum information quantity that module M receives from inputs I_F of flow F and C(I_M) is the total information quantity that module M would receive if isolated.

Similarly, the *observability measure* of module M in a flow F is given by:

$$OB_F(M) \ = \ \frac{T(O_M;O_F)}{C(O_M)}$$

where T(O_F;O_M) is the maximum information quantity that the outputs of flow F may receive from the outputs O_M of module M and C(O_M) is the total information quantity that module M can produce on its outputs.

Controllability (CO) and observability (OB) values of a module are within the interval [0, 1]. If CO (OB) is close to zero, then the module is lack of controllability (resp. observability), and on the contrary the module is controllable (observable) if its CO (resp. OB) is close to one.

The testability measure of module M in flow F, which is a function of the controllability measure and the observability measure, is defined as the couple of values:

$$TE_F(M) = (CO_F(M), OB_F(M))$$

Testability measures computation is more detailed in [12, 13].

3. Automatic generation of the testability model

As our goal is to analyze the testability of the code generated from data-flow designs, particularly avionics software. To apply the SATAN tool, we use the SSA form (Static Single Assignment) [2] to translate code into a data-flow representation. This SSA form has been principally used as a platform for various classical code optimization algorithms in compilation techniques. Then, we build up the testability model from the SSA form.

In this section, we first present the SSA form and then we describe the algorithms which allow the testability model, *i.e.* ITG, to be automatically generated from the SSA form.

3.1. The SSA form

Indeed, the SSA form allows the data flow aspect to be represented from the control flow graph of a program (software). Translating a program into the SSA form is achieved in two steps. In the first step, some special Φ-functions are introduced at each join node in the program's control flow graph. A Φ-function at node X has the form V←Φ(R, S, ...), where V, R, S, ... are variables and the number of operands R, S, ... is the number of the control flow predecessors of X. This Φ-function merges distinct values of a variable to produce a new value according to the control flow. In the second step, the variables R, S, ... are replaced by new variables V_1, V_2, V_3 ... so that each use of V_i is reached by just one assignment to V_i. Indeed, there is only one assignment to V_i in the entire program. The SSA construction is detailed more precisely in [1]. It is implemented in the GCC 3.x compiler.

For example, translating the following function *hcf* (Highest Common Factor) into the SSA form is presented in Figure 5.

```
int
hcf (int a, int b)
{
  while (a != b) {
    if (a > b)
    {
      a = a - b;
    }
    else
    {
      b = b - a;
    }
  }
  return (a);
}
```

(a) Original program

```
int
hcf (int a0, int b0){
  if (a0 != b0) {
    do{
      a1 = Φ(a0, a3);
      b1 = Φ(b0, b3);
      if (a1 > b1) {
        a2 = a1 - b1;
      }
      else {
        b2 = b1 - a1;
      }
      a3 = Φ(a1, a2);
      b3 = Φ(b1, b2);
    }while(a3 != b3);
  }
  a4 = Φ(a0, a3);
  return (a4);
}
```

(b) SSA form

Figure 5. The hcf function and its SSA form

In this example, some Φ-functions are inserted on the join nodes to determine the values for the variables a and b; then the variables are renamed.

Moreover, when translating a program into SSA form, the GCC compiler does not consider pointers. So, we propose some extensions of the program without losing its semantics. For a program, the extensions are composed of two additions: a "prologue" and an "epilogue". At the beginning of the program, an inserted prologue allows replacement of pointers by local variables. In the same way, at the end of the program, an inserted epilogue allows replacement of inserted local variables by pointers.

The indices problem in program analysis is not solved yet in this paper, but we intend to tackle in later work.

The SSA form has two important properties: every use of a variable only depends on a unique definition of this variable; the original program and its SSA form have the same control flow graph. This allows a testability model, which represents control dependencies and data dependencies, to be constructed with the SATAN tool.

3.2. Algorithms of testability model generation

Automatic generation of model testability (ITG) permits the process of testability analysis to be realized as presented in Figure 6.

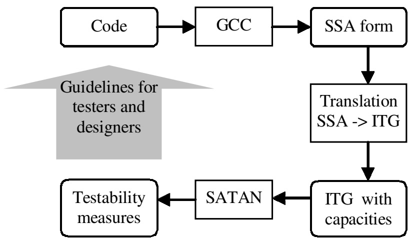

Figure 6. Testability analysis process

The SSA form generated by GCC is represented by a low level language RTL (Register Transfer Language). In order to facilitate the translation into an ITG, we propose the transformation of the SSA form into some intermediate representations. First, the SSA form is transformed into a list of quadruplets, which only contains information needed for the ITG. Then, the list of quadruplets is transformed into a diagram of operations, which is close to ITG, before being translated into the ITG. This is represented in Figure 7.

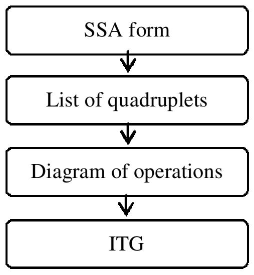

Figure 7. Translating the SSA form into the ITG

Thus, we will define two intermediate representations: the list of quadruplets and the diagram of operations, and we will describe three algorithms: the transformation of the SSA form into a list of quadruplets, the transformation of the list of quadruplets into a diagram of operations, and the construction of the ITG. We will illustrate these algorithms on the above example of function *hcf* (see Figure 5).

Transformation into a list of quadruplets. The SSA form is generated in RTL description. A RTL description is a list of expressions described in a functional language like Lisp. This description is used by the compiler for code optimization, so it contains many details. We only need some information in order to build up the ITG. Thus, we define a list of quadruplets which contains only the data needed for the transformation.

A list of quadruplets is defined as follows:

(list-quadruplets) ≡ ((binary-operator)
(unary-operator)
(assignment)
(definition-of-label)
(conditional-jump)
(unconditional-jump)
(phi))

Each quadruplet is represented by a list, in which the first element is the quadruplet name, and the second is the identification (id) of the quadruplet in the list.

A binary operator (bo) is represented by one operator (op), two operands (l, r), and one result (res):

binary-operator ≡ (bo id (op) (l r) (res))

A unary operator (uo) is represented by one operator (op), one operand (opr), and one result (res):

unary-operator ≡ (uo id (op) (opr) (res))

An assignment (assignt) is represented by one operand (opr) and one result (res):

assignment ≡ (assign id (opd) (res))

A definition of label (label) is represented by the identification of the label (idl):

def-of-label ≡ (label id (idl))

A conditional jump (cj) is represented by one operator (op), two operands (l, r), and one label identification (idl) where the jump reaches:

conditional-jump ≡ (cj id (op) (l r) (idl))

An unconditional jump (uj) only contains the label identification (idl) where the jump reaches:

unconditional-jump ≡ (uj id (idl))

A Φ-function is represented by a sequence of variables (v_1, v_2, . . .) and a result (res):

$\Phi \equiv (\Phi$ id (v_1 v_2 . . .) (res))

Note that, in a list of quadruplets, a sequence of Φ-functions is always preceded by a definition of label, since Φ-functions are introduced at a join node in the control graph of the program.

Transforming the SSA form into a list of quadruplets consists of reading each RTL expression and then extracting the data needed to build up the list of quadruplets. The list of quadruplets of the *hcf* function is given in Figure 8.

Transformation into a diagram of operations. Once the list of quadruplets is constructed, we transform this list into a diagram of operations, in which the data-flow aspect is better expressed. The diagram of operations represents data dependencies between operations, which are composed of primitive operators, function calls, and Φ-functions. In this diagram, the control-flow aspect is hidden in Φ-functions, since every control flow reaches the corresponding Φ-function at a join node in the control flow graph. We represent a diagram of operations by a list of operations as follows:

(diagram-operations) ≡ ((operation))

where each operation is composed of an operator (op), a list of operands (list-opds), and a result (res):

operation ≡ (op (list-opds) (res))

Transforming a list of quadruplets into a diagram of operations consists of searching the quadruplets, which realize operations, in

```
(
    (cj 1 (eq) ($a_0$ $b_0$) (45))
    (label 2 (18))
    ($\Phi$ 3 ($b_3$ $b_0$) ($b_1$))
    ($\Phi$ 4 ($a_3$ $a_0$) ($a_1$))
    (cj 5 (le) ($a_1$ $b_1$) (31))
    (bo 6 (minus) ($a_1$ $b_1$) ($a_2$))
    (uj 7 (13))
    (label 8 (31))
    (bo 9 (minus) ($b_1$ $a_1$) ($b_2$))
    (label 10 (13))
    ($\Phi$ 11 ($b_2$ $b_1$) ($b_3$))
    ($\Phi$ 12 ($a_1$ $a_2$) ($a_3$))
    (cj 13 (ne) ($a_3$ $b_3$) (18))
    (label 14 (45))
    ($\Phi$ 15 ($a_3$ $a_0$) ($a_4$))
)
```

Figure 8. List of quadruplets of the hcf function

order to build up the diagram of operations. We propose a graphical representation of the diagram of operations, in which inputs and outputs are represented by semicircles, operations by rectangles, and variables by circles.

The graphical representation of the diagram of operations of the *hcf* function is presented in Figure 9.

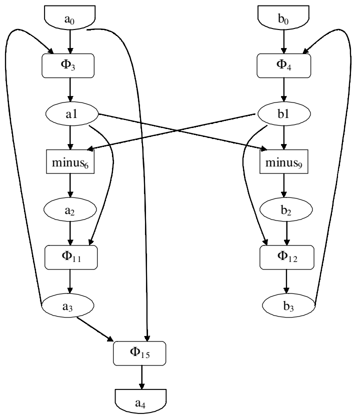

Figure 9. Graphic representation of the diagram of operations of the hcf function

The diagram of operations of the *hcf* function is presented in Figure 10.

Construction of the ITG. The diagram of operations represents the required data-flow aspect, but the control flow aspect is not expressed yet, since it is hidden in the Φ-functions. Hence, we

```
(
    (($\Phi_3$) ($a_0$ $a_3$) ($a_1$))
    (($\Phi_4$) ($b_0$ $b_3$) ($b_1$))
    (($minus_6$) ($a_1$ $b_1$) ($a_2$))
    (($minus_9$) ($b_1$ $a_1$) ($b_2$))
    (($\Phi_{11}$) ($b_2$ $b_1$) ($b_3$))
    (($\Phi_{12}$) ($a_1$ $a_2$) ($a_3$))
    (($\Phi_{15}$) ($a_3$ $a_0$) ($a_4$))
)
```

Figure 10. Diagram of operations of the hcf function

have to analyze the details of the Φ-functions. As we presented in section 3.1, each Φ-function has two or more input variables and one output variable, and the output variable is assigned by one of the input variables according to the control flow. So, a Φ-function is considered as a switch associated with one or several conditions. Once we have all details of Φ-functions in a diagram of operations, we can transform the diagram into the corresponding ITG. The construction of the ITG is the following:

```
Construction-of-ITG
    Analyzing-Φ-functions
    Transforming-into-ITG
```

Analyzing Φ-functions aims at looking for the conditions associated with each Φ-function and then assigning each input variable of a Φ-function to each branch of switch. This phase is as follows:

```
Analyzing-Φ-functions
    For eachΦ-function
        Looking-for-conditions-associated-with-Φ
        Assigning-input-variables-to-branches
```

To look for the conditions associated with a Φfunction, we have to analyze all jumps and labels in the list of quadruplets, which are related to the Φ-function. In this paper, we only deal with the if-then-else statement, *i.e.* Φ-function with two variables. This analysis proceeds as follows:

```
Looking-for-conditions-associated-with-Φ
    Taking the label (L1) preceding the
    Φ-function in the list of quadruplets.
    Looking for the jump quadruplet (j)
    reaching the L1 label.
    If jump j is an unconditional jump
        Taking the label (L2) of the following
        quadruplet in the list of quadruplets.
        Looking for the conditional jump
        associated with the L2 label.
        We obtain the condition.
    If jump j is a conditional jump
        We obtain the condition.
```

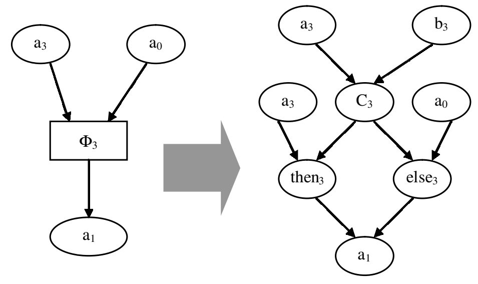

Figure 11. Graphical representation of a Φ-function

For example, analyzing the Φ_3 function in the diagram of operations of the *hcf* function is to transform the Φ_3 operation into the then-else in Figure 11.

In this example, the condition associated with Φ_3 is determined by the two operands a_3 and b_3, and the condition code C_3 (operator "! =").

Then, the diagram of Φ_8 in Figure 11 can be easily transformed into an ITG by inserting transitions in Figure 12.

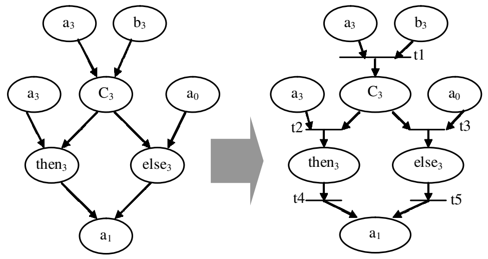

Figure 12. Transforming a Φ-function into an ITG

As the transitions allow the information transfer between the places to represented, each transition is inserted when there is an information transfer between places. In Figure12, the $t1$ transition is added because the module C_3 needs the information from inputs a_3 and b_3. The $t2$ and $t3$ transitions are inserted for the similar reason. The $t4$ and $t5$ are inserted because the module a_1 needs information from either the module *then*$_3$ or the module *else*$_3$.

When all Φ-functions in the diagram of operations are analyzed, the control flow aspect is explicitly expressed. Now, we can transform the diagram of operations into an ITG by only adding transitions between nodes. The ITG of the *hcf* function is presented in Figure 13.

4. Case study

In our case study, some data-flow designs and the code generated from these designs were provided by THALES Avionics. In this paper, the study is concerned with a piece of avionics software called TIT. The TIT design is composed of 19 components, which are connected between them. The code is generated by the GALA tool in C language. In fact, components used in a design are implemented in a library, and GALA generates the code by gathering the code of the components. The generated code of TIT contains 21 variables and 47 operators. It has 258 lines of code. The cyclomatic number of TIT code is 9.

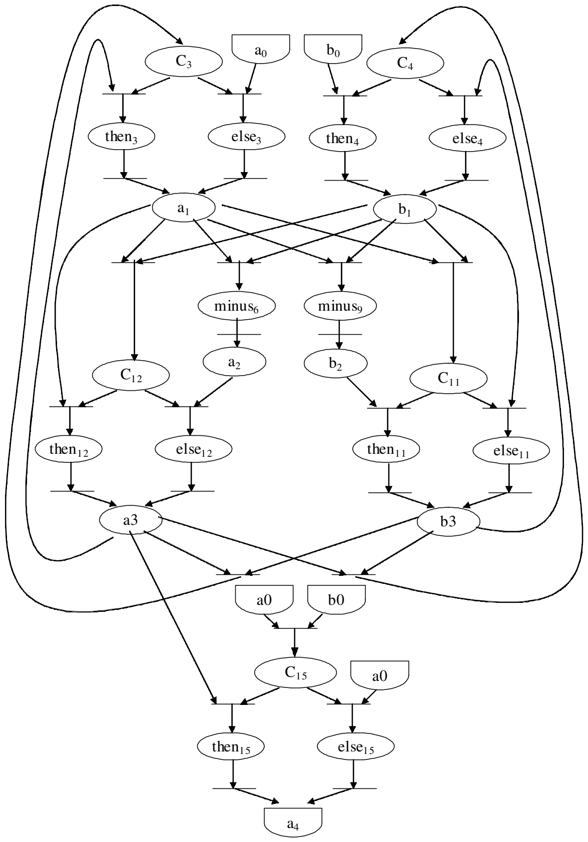

Figure 13. The ITG of the hcf function

This study's goal is to determine the testability of the automatically generated code. Our main objective is to recommend whether design should be modified to produce a code of higher testability. We have not yet completed those recommendations; we have only found the testability for the code generated.

We first generate the SSA form of the TIT code with GCC. Then, the ITG is automatically built up from its SSA form. Finally, the SATAN tool computes the testability measures based on the ITG.

The generated ITG of TIT code is composed of 72 modules and 86 transitions. The set of flows computed by SATAN contains 178 flows. Hence, if we want to test the TIT code by covering all paths, i.e. flows in the ITG, then the number of test cases is quite important. Such a test strategy is costly in terms of time. However, the testing goal is also related to the diagnostic. As said in section 2, in the case of simple faults (in maintenance phase) we apply the "Multiple-Clue" strategy to choose a minimum number of flows that cover all modules in ITG. On the contrary, in the case of multiple faults (in development phase), the "Start-Small" strategy is more effective. When applied to the TIT code, the "Start-Small" strategy identifies 13 flows for testing, and the "Multiple-Clue" strategy identifies 9 flows. So, we can state that the number of flows needed to cover all modules in the ITG is considerably reduced.

The testability measures computed for the TIT are given in Table 1 and Table 2.

Table 1. Controllability values

Controllability values	1.0	0.9
Number of modules	71	1

Table 2. Observability values

Observability values	1.0	0.81	**0.08**	**0.06**
Number of modules	23	17	**4**	**28**

According to these results, we can state that modules controllability are acceptable, but there are 32 modules with bad observability values. Then, some observation points should be added in order to improve modules observability.

5. Conclusion

The paper has presented an approach of testability analysis of a code automatically generated from data-flow designs. The use of the SSA form allows code testability to be analyzed with our SATAN tool. The analysis at code level is more complex than at the design level, but it gives more accurate results. Particularly, this approach helps to analyze testability of some embedded data-flow software code before using it, what is required by certification authorities.

Since the SATAN method can analyze testability of data-flow software, at the design level as well as at the code level, thus testability analysis can be considered throughout the entire software development project.

In future work, we intend to extend the approach to analyze testability of any program code in imperative languages.

References

[1] P. Briggs, T. Harvey, and T. Simpson. Static single assignment construction. Technical report, May 1995.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wergman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transaction on Programming Languages and Systems*, 13(4), Oct. 1991.

[3] R. S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, June 1991.

[4] S. Jungmayr. In *Proceedings of 12th International Workshop on Software Measurement*, Magdeburg, Gemany, 2002.

[5] K. Karoui and R. Dssouli. Specification transformations and design for testability. In *Proceedings of IEEE Global Telecomminucations Conference GLOBECOM'96*, pages 680–685, London, England.

[6] Y. Le Traon and C. Robach. Towards a unified approach to the testability of co-design system. In *Proceedings of IEEE International Symposium on Software Reliability Engineering*, pages 278–285, Toulouse, France.

[7] Y. Le Traon and C. Robach. Testability measurements for data flow design. In *Proceedings of the Fourth International Software Metrics Symposium*, Albuquerque, New Mexico, Nov. 1997.

[8] T. J. McCabe. A complexity measure. *IEEE Transactions On Software Engineering*, SE-2(4):308–320, Dec. 1976.

[9] B. A. Nejmeh. Npath: A complexity measure of execution path complexity and its applications. *Communications of the ACM*, 31(2):188–200, Feb. 1988.

[10] T. Nguyen, M. Delaunay, and C. Robach. Testability analysis for software components. In *Proceedings of the International Conference on Software Maintenance*, pages 422–429, Montréal, Canada, October 2002.

[11] Petrenko, R. Dssouli, and H. Koenig. On evaluation of testability of protocol structures. In *Proceedings of the Intenational Workshop on Protocol Test Systems (IFIP)*, pages 111–123, Pau, France, 1993.

[12] C. Robach and S. Guibert. Information based testability measures. In *Proceedings of Silicon Design Conference*, pages 429–438, Wembley, G.B., 1986.

[13] C. Robach and S. Guibert. Testability measures: a review. *International Journal of Computer Systems Science and Engineering*, 3:117–126, 1988.

[14] J. M. Voas and K. W. Miller. Semantic metrics for software testability. *Journal of Systems and Software*, 20(3):207–216, Mar. 1993.

[15] J. M. Voas and K. W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, May 1995.