Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 116 (2005) 213–225
www.elsevier.com/locate/entcs

Testability Analysis of Data-Flow Software

Thanh Binh Nguyen, Michel Delaunay, Chantal Robach [1]

LCIS-ESISAR, 50, rue Barthélémy de Laffemas, Valence, 26902, France

Abstract

This paper is about testability analysis for data-flow software. We describe an application of the SATAN method, which allows testability of data-flow designs to be measured, to analyze testability of the source code of critical data-flow software, such as avionics software. We first propose the transformation of the source code generated from data-flow designs into the Static Single Assignment (SSA) form; then we describe the algorithm to automatically translate the SSA form into a testability model. Thus, analyzing the testability model can allow the detection of the software parts which induce a testability weakness.

Keywords: Program analysis, testability measures, data-flow software.

1 Introduction

In the software development process, the validation and verification phases play an important role, in which testing is a crucial task. During this task, faults must be revealed. However, testing increases the reliability of the software, it never can ensure that there are no faults in the software. In addition, when the software is rather complex, the testing task is time consuming and highly costly. This is why some testability metrics have been studied these last years in order to help in appraising the ease/difficulty for testing software.

In this paper, we focus on testability analysis of a source code generated from data-flow designs. This analysis can be used:

- to identify parts of a design/code which contribute to a lack of testability,

[1] Email: {Binh.Nguyen-Thanh, Michel.Delaunay, Chantal.Robach}@esisar.inpg.fr

doi:10.1016/j.entcs.2004.02.078

- to help designers and testers to distribute resources during the design phase and the testing phase,
- to compare the design testability and the generated code testability in order to evaluate the coding process,
- to improve the software reliability.

A testability analysis is very essential for critical software in order to reduce testing cost and insure software quality; as these software parts are often designed with a data-flow approach, this paper is concerned with data-flow software.

In a previous work, Le Traon and Robach [6] proposed a testability model, which is implemented in the SATAN tool (System's Automatic Testability ANalysis), to analyze testability of data-flow designs. 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 these designs or component code used in these designs: because the source code is often generated or described in imperative languages like C or ADA. So, we recently proposed the use of the Static Single Assignment (SSA) form to apply this model for testability analysis of component code [8]. 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 an algorithm to automatically translate the SSA form into the testability model, which is used to compute the testability measures for the source code.

The paper is organized as follows. Section 2 is about some related work. Section 3 briefly presents the principles of the SATAN tool which allows the testability analysis of data-flow designs. Section 4 presents the extension of the SATAN tool to analyze code testability. An algorithm is described in Section 5 to automatically compute testability measures. Section 6 presents a case study. Finally, we give our conclusion in Section 7.

2 Related work

Several research works about software testability have been published. Each one of them was investigated in order to analyze testability within a specific application area.

Freedman [3] proposed the testability measures for non-deterministic 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 [13] also proposed testability metrics based on the input and output domains of a software component. Both these methods allow the measurement of testability at component level by analyzing functional specifications; they can be used to rank components with respect to testability.

Voas and Miller proposed the PIE (Propagation, Infection and Execution) technique to analyze software testability in [14]; this technique measures testability of each statement in software by a dynamic analysis, *i.e.* while running the software. Some software complexity measures were also considered as a substitute for testability measures, such as McCabe's metrics (cyclomatic number) [7] or Nejmeh's metrics (NPATH) [10]. All these methods proposed by Voas and Miller, McCabe, and Nejmeh can only be applied to source code.

Jungmayr [4] presented testability measurement in the context of static dependencies within object-oriented software.

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.

Le Traon and Robach [6] proposed testability measures particularly for data-flow designs, which are graphically represented by diagrams of components.

3 SATAN tool

In this section, we present some principles of the SATAN tool to analyze testability of data-flow designs [6].

In the SATAN tool, *testability* of software is based on the controllability and the observability of the software components. *Controllability* is defined as the ease to forward data from the inputs of the software to the inputs of a component. *Observability* is defined as the ease to propagate data from the outputs of a component to the observable outputs of the software.

The testability analysis is founded on a functional model, which 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 presentation of the functional model on the example of a data-flow design in Figure 1.

In this diagram, $o1$ and $o2$ are a couple of outputs, $o1$ is a boolean output used 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

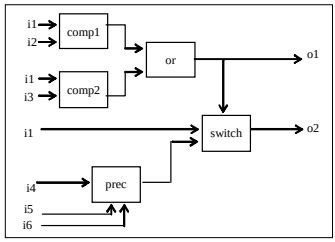

Fig. 1. A data-flow design

component.

3.1 Functional model

The principle of modeling the information transfers through software consists of representing the control and data-flow aspects on a same graph that is called the *Information Transfer Graph* (ITG). It is a bipartite directed graph without cycles. 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):

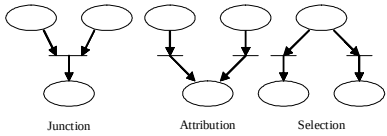

Fig. 2. Information transfer modes

- 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.

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.

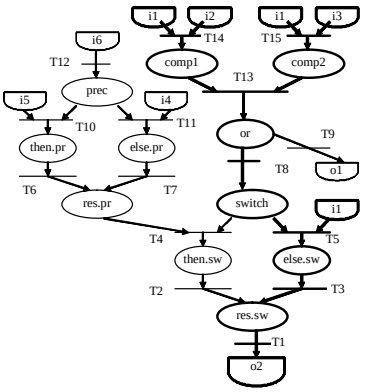

Fig. 3. Information Transfer Graph

Note that in an ITG a cycle (if it exists) is assumed to be exercised once.

3.2 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 or some outputs. It contains a set of places, transitions, and edges. Thus, a flow can be considered as a sub-graph; 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.

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.

3.3 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: progressive structural strategy (*Start-Small*) and cross-checking strategy (*Multiple-Clue*). The cross-checking strategy is based on choosing a subset of flows that satisfy coverage of all the modules: all chosen flows 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 progressive structural 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). Moreover, we recently proposed an improvement of the effectiveness of both these strategies by using some accessibility measures in [9].

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.

3.4 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 measure estimates the information quantity available on the outputs of the software from the outputs of a module. This principle is illustrated in Figure 4.

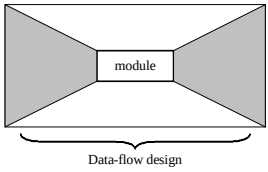

Fig. 4. Controllability and Observability of a module

Moreover, 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.

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.

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].

4 Code testability analysis

As our goal is to apply the SATAN tool for analyzing the testability of the code source, 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. The testability analysis is based on the SSA form.

In this section, we first present the SSA form and then we describe the process allowing automatic testability analysis of source code.

4.1 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 *delta* into the SSA form is presented in Figure 5.

```
int delta (int a, int b)
{
        if (a > b)
                a = a – b;
        else
                b = b – a;
        return (a*b);
}
```

(a) Original program

```
int delta (int a0, int b0)
{
        if (a0 > b0)
                a1 = a0 – b0;
        else
                b1 = b0 – a0;
        a2 = ϕ(a0, a1);
        b2 = ϕ(b0, b1);
        return (a2*b2);
}
```

(b) SSA form

Fig. 5. The *delta* function and its SSA form

In this example, two Φ-functions are inserted on the join nodes to determine the values for 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 SSA form has some 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; and the original program has the same semantics than its SSA form.

4.2 Analysis process

From a SSA form we can construct the corresponding ITG. Then, to use the SATAN tool to compute testability measures, *i.e.* controllability and observability measures, we must associate each module of ITG with a capacity. This module capacity allows appraising the information loss through the module. An ITG with module capacities is called Information Transfer Net (ITN).

We propose a process of testability analysis as presented in Figure 6.

In this process, a C program is first translated into the SSA form by the GCC compiler. Then, the SSA form is translated into the ITG. Each module of ITG is associated with its capacity to produce the ITN. Finally, the SATAN tool computes testability measures from the ITN.

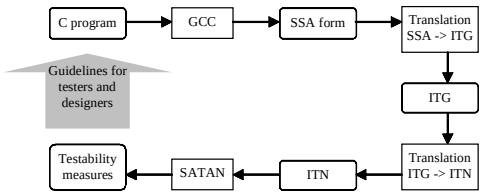

Fig. 6. Testability analysis process

5 Construction of the functional model

In this section, we briefly describe how to translate the SSA form into an ITG, then an ITN.

5.1 Translation into an ITG

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 representation. First, the SSA form is transformed into a list of quadruples, which only contains information needed for the ITG. Then, the list of quadruples is transformed into a diagram of operators, which is close to the ITG, before being translated into the ITG. This translator is given in Pascal pseudo-code as follows:

```
Translator-SSA-ITG(SSA-form)
    list-of-quadruples ← transformation-into-quadruples(SSA-form)
    diagram-of-operators ← transformation-into-operators(list-of-quadruples)
    ITG ← construction-ITG(diagram-of-operators)
```

5.2 Translation into an ITN

Once we obtain an ITG from the SSA form, we need to transform the ITG into an ITN by adding all module capacities. As we told above, a module capacity is the information quantity that is available on the module outputs from its inputs. So, we first determine the types of the inputs and outputs of each module in ITG. Then, capacities will be evaluated for all the modules. Finally, the ITN is constructed from the ITG and the module capacities. This translator is given as follows:

```
Translator-ITG-ITN(SSA-form, ITG)
    types-of-inputs-outputs ← determination-types-of-inputs-outputs(SSA-form)
    capacities-of-modules ← computation-capacities(types-of-inputs-outputs)
    ITN ← construction-ITN(ITG, capacities-of-modules)
```

They are implemented in Lisp language and they are integrated in the SATAN tool.

6 Case study

We show in this section the application on a data-flow diagram provided by THALES Avionics, then we also apply our approach to the code generated by the GALA tool[2] from this diagram. The diagram, called *sub-THT*, is given in Figure 7.

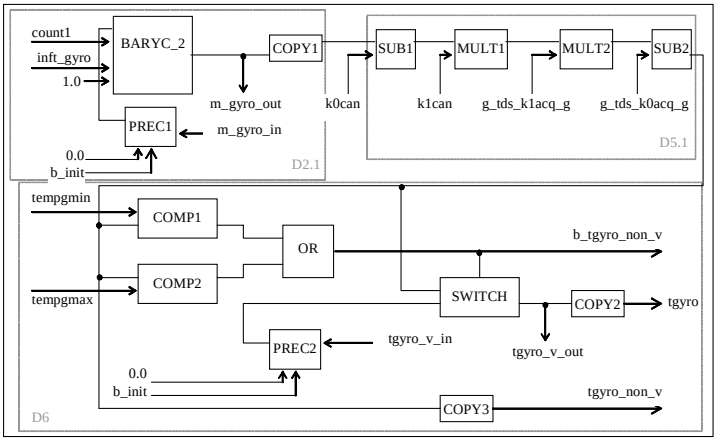

Fig. 7. Diagram *sub-THT* for the case study

In this diagram, $BARYC$ is a computation component; $COPY_i$ is to assign a value to an output; $COMP_i$ is a comparison component; OR is a logical component; SUB_i and $MULT_j$ are respectively the subtraction and multiplication components; $SWITCH$ is a selection component; and $PREC_i$ is a memorization component. The diagram has several inputs and outputs. In particular, two outputs $b\_tgyro\_non\_v$ and $tgyro\_non\_v$ form a couple: the boolean value of $b\_tgyro\_non\_v$ determines whether the value of $tgyro\_non\_v$ is valid. This diagram is designed for a piece of the avionics software used at THALES Avionics.

We first analyze the diagram and then the code generated from this diagram.

6.1 Diagram analysis

The ITG is directly constructed from the diagram. It is given in Figure 8.

[2] The GALA tool is developed by THALES Avionics to specify and generate avionics software

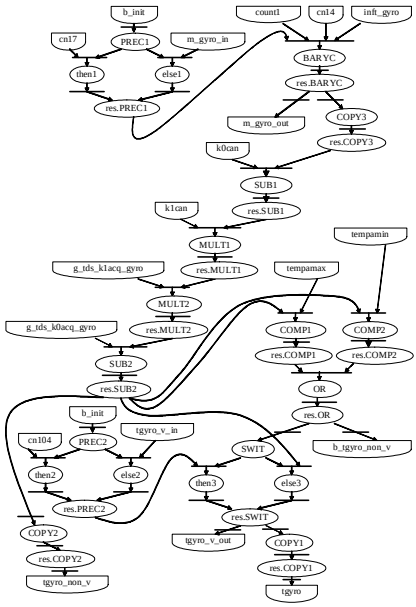

Fig. 8. The ITG of diagram *sub-THT*

In this ITG, the SATAN tool identifies 12 flows. Then it also computes testability measures of each module in each flow. One module can have different values of testability in different flows. So, a fine analysis can be done by examining testability of each module in each flow, then all modules with low testability will be identified. The testability of such the modules should be improved, or they must be carefully tested.

However, in this paper, we only give the testability of the least testable module in each flow, because this module is the most difficult for testing in the flow. The testability (controllability, observability) of this module can stand for the testability of the flow. In this case, among 12 flows, 6 flows have the maximal testability (1.0, 1.0), and 6 others have the maximal controllability but the low observability (1.0, 0.0833). So, the observability of some modules in the last 6 flows should be improved.

6.2 Code analysis

We now analyze the code generated by the GALA tool from diagram *sub-THT*. As the code contains more details than the diagram, so the ITG obtained from the code contains more elements (modules, transitions, edges) than the one obtained from the diagram. Here, we do not present the ITG because of limited presentation space. The SATAN tool identifies 42 flows in the ITG. The number of flows from code analysis is increased with respect to the number

of flows from diagram analysis. As the ITG is more detailled, the flows are more numerous. Table 1 gives the information for the two levels of analysis.

Table 1
Information on ITG for the two levels of analysis

Analysis	Number of modules	Number of transitions	Number of flows
Diagram	36	104	12
Code	48	134	42

Relating to measures, among 42 flows, 20 flows have the testability (1.0, 1.0), and 22 others have the testability (1.0, 0.0833). Comparing the measures obtained from two levels of analysis, we state that they are very similar. Hence, we can say the coding process does not have impact on the testability. However, some modules should be reviewed to improve the observability measures.

7 Conclusion

The paper has presented an approach of testability analysis of generated code source as well as data-flow designs. The use of the SSA form allows code testability to be analyzed with our SATAN tool, which was used successfully to analyze testability of data-flow designs. The analysis at code level is more complex than at the design level, but it gives more detailed measures about testability. Particularly, this approach helps to analyze testability of avionics software before using it, which is required by certification authorities. Moreover, code testability analysis can help to identify the low testable parts in software re-engineering.

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.

References

[1] Briggs, P., T. Harvey, and T. Simpson, *Static Single Assignment Construction*, Technical report, Rice University, 1995.

[2] Cytron, R., 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** (1991).

[3] Freedman, R. S., *Testability of Software Components*, IEEE Transactions on Software Engineering, **17** (1991), 553–564.

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

[5] Karoui, K., and R. Dssouli, *Testability Analysis of the Communication Protocols Modeled by Relations*, Technical report, No. 1050, Département d'Informatique et de Recherche Opérationnelle, Faculté des Arts et des Sciences, Université de Montréal, 1996.

[6] Le Traon, Y., and C. Robach, *Testability Measurements for Data Flow Design*, Proceedings of the Fourth International Software Metrics Symposium, Albuquerque, New Mexico, 91–98, 1997.

[7] McCabe, T. J., *A Complexity Measure*, IEEE Transactions On Software Engineering, **SE-2** (1976), 308–320.

[8] Nguyen, T. B., M. Delaunay, and C. Robach, *Testability Analysis for Software Components*, Proceedings of the IEEE International Conference on Software Maintenance, Montréal, Canada, 2002, 422–429.

[9] Nguyen, T. B., M. Delaunay, and C. Robach, *Testing Criteria for Data Flow Software*, Proceedings of the 10th Asia-Pacific Software Engineering Conference, Chiang Mai, Thailand, 2003, 332–339.

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

[11] Petrenko, A., R. Dssouli and H. Koenig, *On Evaluation of Testability of Protocol Structures*, Proceedings of the Intenational Workshop on Protocol Test Systems (IFIP), Pau, France, 1993.

[12] Robach, C., and S. Guibert, *Information Based Testability Measures*, Proceedings of Silicon Design Conference, 429–438, Wembley, GB, 1986.

[13] Voas, J. M., and K. W. Miller, *Semantic Metrics for Software Testability*, Journal of Systems and Software, **20** (1993), 207–216.

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