Structural Analysis of Explicit Fault-Tolerant Programs

S. Gossens, M. Dal Cin
Informatik 3
Universität Erlangen-Nürnberg
Germany
{gossens,dalcin}@informatik.uni-erlangen.de

Abstract:

Explicit fault tolerant programs are characterized by proactive efforts to ensure robustness and ability of fault correction. A fault tolerant application is usually realized conforming to one of a collection of standard techniques. Graph based methods can be used to examine existing applications to derive a control flow abstraction with respect to the fault-tolerance architecture. This abstraction, which we call the fault tolerance behavioural type*, can be used as basis of structural analysis of the implemented architecture. This paper outlines the basic ideas and demonstrates their application using CTL model checking to verify fault tolerance properties of explicit fault-tolerant programs.*

Key Words:
fault tolerant programming, analysis of fault tolerance features, explicit fault tolerance, control flow abstraction, model checking

1. Introduction

It is commonly agreed upon that in complex software systems it has become infeasible to completely avoid software design faults. Therefore, key abilities of software systems in safety critical application domains are their robustness as well as their tolerance against design faults. Fortunately, some degree of robustness and tolerance against design faults can be provided by replication of important processing components (protective redundancy). Independently calculated results of these redundant components are used in voting and correction components.

We distinguish explicit fault tolerance from transparent fault tolerance. In explicit fault tolerance, the application software actively provides for fault tolerance, while in transparent fault tolerance, the infrastructure (e.g. middleware or operating system) realizes fault tolerance measures without cooperation with the application [6, 10, 15]. While in transparent fault tolerance, the fault tolerant components remain transparent to the application programmer, they are implemented in explicit fault tolerant programs' structures and thus potential subject to fine tuning and analysis.

In this paper, we outline a methodology for the structural analysis of explicitly fault-tolerant programs with respect to their imbedded fault tolerance features. The paper is organized as follows: First, we overview explicit fault tolerance and a method of structural analysis, behavioural typing, in sections 2 and 3. Then, we show how to use these techniques to verify the existence of fault-tolerant program structures in sections 4 and 5, additionally employing CTL model checking.

2. Explicit fault-tolerant programs

Fault tolerance features of a program correspond to characteristic patterns of the control flow. E.g., in a system that can detect a single failure, there should not only be a twofold calculation of the critical values, but also a comparison of these before use or output in every possible flow of control. Structural characteristics of this kind offer the basis for verifying or reengineering software with the goal to determine and classify robustness and

fault tolerance features.

2.1 Explicit Fault tolerance

Several different techniques have been developed for explicit software fault tolerance. The two prominent ones are – N-version or Multiversion Programming (MVP) [1-4, 8, 11] and Recovery Blocks (RB) [10, 12]. A third, related technique is N self-checking programming [9].These techniques incorporate systematically dedicated redundancy into software modules.

RB is a language construct supporting the incorporation of program redundancy in a concise form. The structure of recovery blocks is as follows:

```
module xy { RECOVERY BLOCK
                ensure AT
                by B1
                else by B2
                ….. ….
                else by Bn
                else error handler
          }
```

Here AT denotes an acceptance test, B_1 a primary program block and B_n , n>1, alternative blocks. The blocks are executed sequentially, and the acceptance test is used to check the results. The alternative blocks are executed only if the results of the current block fail the acceptance test (after undoing the effects of the previous block). Recovery blocks can be nested.

In MVP, different versions of software modules are executed, and the results of these versions are voted on. A commonly used syntax is as follows:

```
module xy{ MVP BLOCK
               B1,
               B2,
               B3
               VOTER voteproc
               error handler
         }
```

Here, B_i are program blocks the results of which are voted on by procedure *voteproc*.

Thus, RB employs time redundancy and NVP masking redundancy. Both techniques are based on design diversity. They are effective only if the different software versions are independent witch respect to fault generation. However, if the alternative blocks are distributed on different hardware, this redundancy technique can also be used to tolerate hardware faults [6, 14]. Then there is no need for diversity.

Architectural descriptions of fault tolerant programs can be nested and take more complex shapes. The following shows a simple example of a fault-tolerant software module where RB and MVP schemes are nested. Here, the recovery block serves as a the error handler of the multi version programming scheme:

```
module nvsort{ MVP BLOCK
                  quicksort(in A, out C)
                  shellsort(in A, out D)
                  insertionsort(in A, out E)
                VOTER
                 majority(in C, D,E , out B)
                   module rbsort{
                       RECOVERY BLOCK
                           ensure A[j+1]>=A[j]
                                for j = 1 to n-1
                         by quicksort(in A, out B)
                         else by shellsort(in A, out B)
                         else by insertionsort(in A, out B)
                         else printError(no sort )
                   }
     }
```

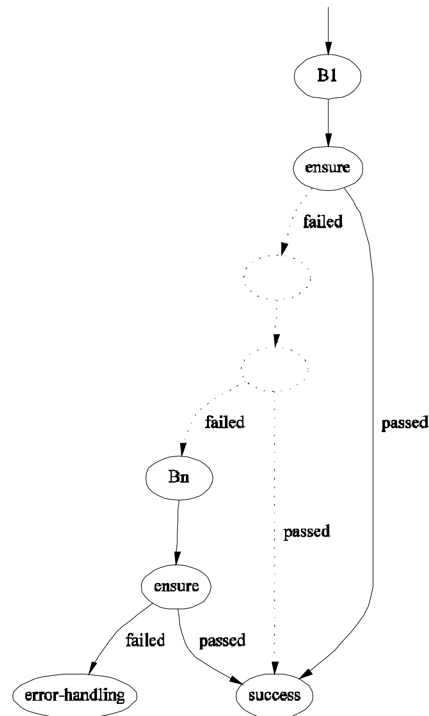
Explicit fault tolerant programming is usually employed for critical software components embedded in a larger program. Having such a possibility is important for practical purposes, since for large software systems it may not be feasible or economically meaningful to provide explicit software fault tolerance for the entire system. However, embedding fault tolerant modules in a system introduces the issue of verifying their fault tolerance features and maintaining their consistency. This holds particularly for legacy software. Therefore, a method to analyze the structure of the entire software system with respect to the embedded fault tolerance mechanisms is highly desirable.

3. Structural Analysis

3.1 Control Flows

Every software system conveys its own unique control flow. This software structure can be analyzed in search for known patterns of fault tolerance like multi version programming, recovery block scheme and the like. If, for example, we have to ensure that a calculated value may never be used without previous voting (like in MVP), this feature can be proven by the absence of a corresponding flow in the program's fault handling.

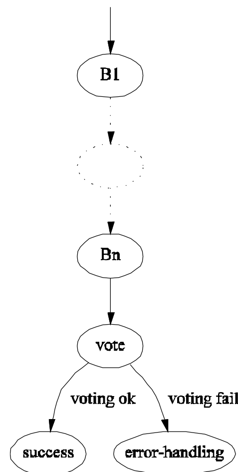
The explicit fault tolerance of a software system is defined by substructures of the control flow graph consisting of the elements ensuring fault tolerance (voters, checking functions etc.) and the possible flows between them. These substructures can be extracted by control flow abstraction [2], resulting in special views, or *fault tolerance behavioural patterns*, of a control flow. They are compact, interest-focussed versions of control flow graphs that directly correspond to the fault tolerance structure of a software. We will discuss them in the following section.



Recovery Block Scheme

3.2 Behavioural Patterns

Fault tolerance architectures are mirrored in their implementations' control flows.
The following Figures represent the control flows conveyed with recovery block style design and multiversion programming.

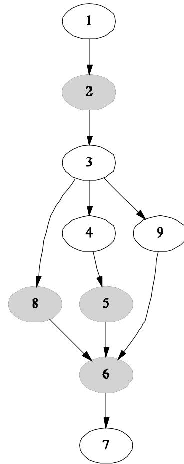


Multiversion Programming

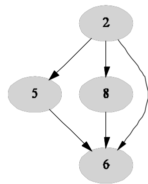
Characteristic control flow schemes like these are embedded in every implementation using special explicit fault tolerance techniques. Therefore, the implementations' control flow graphs contain structural information about the used fault tolerance technique. However, we will hardly find simple and 'clean' control flow graphs like those presented. In reality, a control flow graph will contain many more elements than the architectural components for fault tolerance, and fault-tolerant control flow structures and elements may be mixed and nested with non-fault-tolerant structures and elements.

However, we can abstract given control flow graphs to an architectural view using a technique we call *behavioural typing* [7]. In this technique we select a certain set of vertex types from the control flow graph and calculate the reachability among these vertices. The resulting reachability relation, seen as a relational graph, represents the source graph's *behavioural type* with respect to the selected

set of vertices. Behavioural types have the property of greatly simplifying complex graphs to focus on desired substructures. The following figure illustrates the concept.



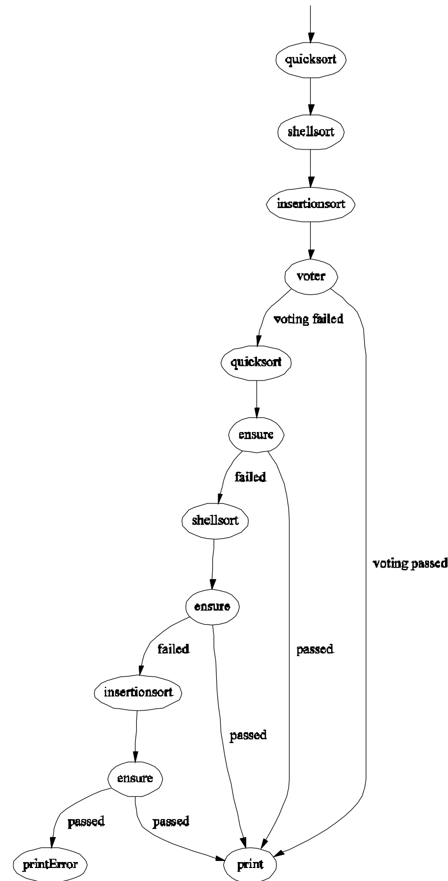
Source Graph



Behavioural Type with Respect to Vertices 2, 5, 6, 8

If we know the elements (statements, functions etc.) enabling fault tolerance that may be contained in a control flow graph, we can choose them as the filtering vertex set and result in behavioural types with respect to fault tolerance, or *fault tolerance (ft)-behavioural types*. Actually, the graphs given for the MVP and RB structures above already represent such ft-behavioural types for the corresponding architectures.

Ft-behavioural types need neither be simple nor follow a single pattern of fault tolerance. Consider again the more complex example MVP-RB architecture of section 2.1. The following figure shows a ft-behavioural type associated with this nested fault-tolerant design:



Ft-Behavioural Type of Example Program

Analyzing a system's ft-behavioural type, we can classify its fault tolerance design based on the ft-behavioural type's graph structure. This gives both a method of finding fundamental implementation flaws as well as a means of reengineering legacy software with respect to its fault tolerance, by classifying ft-behavioural types according to reference architectures. To do this directly, we have to solve problems of graph matching with their respective high computational complexity. However, if we succeed in describing fault tolerance architectures, or at least important characteristics of them, by propositions over a ft-behavioural type, this opens the path to other possibilities of analysis, e.g. direct path checking by traversal algorithms or model checking.

4. Path Searching

Control flow structures can be searched via graph traversal algorithms to provide information about the structure. For example, a breadth-first search can easily provide information upon the existence of a certain path or type of path. However, the approach is non-versatile, as a dedicated program has to be formulated for every type of path to be found. To remain more general and flexible, we can employ path description languages and their associated algorithms to recognize paths in given graphs. It has become popular to use temporal logic formulae to describe paths, or rather properties associated with paths. The techniques to recognize them in given graphs are known as *model checking* [5]. We show how to employ computation tree logic (CTL) model checking to check for desired structural fault tolerance properties below.

4.1 Model Checking

Any directed graph can easily be interpreted as a Kripke structure which forms the basis for CTL model checking. A Kripke structure is a triple M=(S,R,Label) where S is a non-empty set of states, R is a total relation on S which relates the Elements of S to possible successor states, and Label is a function that relates atomic propositions to the members of S. For control flow graphs this means, that every statement in a program represents a state (that is, S contains a vertex for every occurrence of a statement), and R relates every member of S to the vertices of the statements that may be next in control flow. The atomic propositions we will be checking are identifiers associated with certain statements. Thus, Label relates to every statement in a program an identifier (e.g. "PRINT", "IF", or e.g. "quicksort()" for a function call). A Kripke structure based on ft-behavioural types, therefore, contains solely atomic propositions that correspond to statements providing fault tolerance.

As described, we can assume characteristic path structures for every fault tolerant architecture. These path structures can be described in CTL. E.g., the most important feature of an MVP design is that there is no path in the program that uses an item of fault tolerance without previous voting. Knowing that our Kripke structure contains only states that are labeled with exactly one positive atomic proposition, we can express this with the simple formula

¬**E**[¬VOTER **U** PRINT]

We will use this CTL property to demonstrate the approach in the following section.

5. A Case Study

We use the analyzer tool mcana [7] to generate ft-behavioural types from Motorola HC05 microcontroller assembler programs and the CTL model checker SMV [16] to verify path assertions on them. First, mcana generates control flow graphs and abstracts ft-behavioural types. We use atomic propositions tailored to the desired property, focussed on voting and displaying (leaving out calculations in the ft-behavioural type to keep the example presentation handy). The ft-behavioural type is converted to an SMV MODULE description and checked against the formula given in section 4. Verifying the formula with no resulting counterexample path indicates that the desired property holds, i.e. that the corresponding error is absent, while a returned counterexample stands for an error in the fault tolerance control flow.

Consider the small assembler program fragment below. It contains code to take measurements from an external source, multiply them by two with MVP protection, and finally display the result (or an error, if the multiplication was faulty and could not be corrected by the fault tolerance measures). We omit the complete code here for space reasons and focus on the parts relevant for fault tolerance.

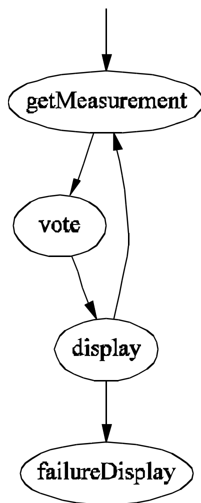
```
              ORG    RAM
;Declaration of Variables

voteValueA    RMB 1 ;address of sort result
voteValueB    RMB 1 ;address of sort result
voteValueC    RMB 1 ;address of sort result
Temp          RMB 1 ;temporary variable

measurementResult
              RMB 1 ;result of measurement
```

```
                       ORG    ROM
;Program (and constants)
start:
          jsr getMeasurement
          sta measurementResult  ; store
measured value in RAM
          jsr multiplyByTwo1
          sta voteValueA
          lda measurementResult
          jsr multiplyByTwo2
          sta voteValueB
          lda measurementResult
          jsr multiplyByTwo3
          sta voteValueC
              ; set  cursor to pos 7 row 2
          lda #$80+$47
          jsr sendcom
          jsr voter
finish:
          jsr display
             ; check whether voting failed
          beq failure
          jmp start
failure:
          jsr failureDisplay
stuck: jmp stuck

multiplyByTwo1:
; use cpu multiplier for arithmetid
          ldx #2
          mul          rts

multiplyByTwo2:
; use cpu shifter for arithmetic
          asla
          rts

multiplyByTwo3:
; use cpu adder for arithmetic
          stx Temp
          add Temp
          rts
voter:
          lda voteValueA
          cmp voteValueB
          beq finishVoting
          cmp voteValueC
          beq finishVoting
          lda voteValueB
          cmp voteValueC
          beq finishVoting
          lda #0 ; indicate error
finishVoting:
          rts
```

This rather small example already corresponds to a control flow graph of considerable size and complexity (although some subroutines' codes are already hidden):

Control Flow Graph of Assembler Fragment

The mcana analyzer has been configured to calculate the structures (ft-behavioural types) between *getMeasurement*, *vote*, *display*, and *failureDisplay* calls, as these are the operations (vertices) relevant to fault-tolerant structures. The ft-behavioural type abstracted by mcana from the control flow graph is shown below:



This ft-behavioral type, together with the property to be proven, corresponds to the following SMV input (note that the terminating state *failureDisplay* has to be affixed with a self-loop to maintain totality of the Kripke structure's successor relation):

```
MODULE main

VAR
   cfg: { getMeasurement, vote, display,
failureDisplay };

ASSIGN
   init( cfg ) := getMeasurement;
   next( cfg ) := case
                  cfg = getMeasurement :
vote;
                  cfg = vote : display;
                  cfg = display : {
getMeasurement, failureDisplay };
                  cfg = failureDisplay :
failureDisplay;
               esac;

SPEC
   !E[ cfg != vote U cfg = display ]
```

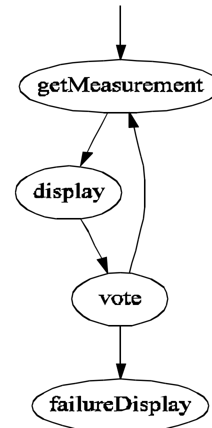
Already by looking at the ft-behavioural type, it can easily be recognized that this ft-behavioural type fulfills the desired property of not printing (using the *display*-routine) before voting. Thus, SMV reports

```
-- specification !E[ cfg != vote U cfg =
display ] is true
```

Certain design faults are reflected directly by the ft-behavioural type. Consider the following malign alteration of the main routine:

```
start:
          jsr getMeasurement
          sta measurementResult
          jsr timesTwo1
          sta voteValueA
          lda measurementResult
          jsr timesTwo2
          sta voteValueB
          lda measurementResult
          jsr timesTwo3
          sta voteValueC
             ; set cursor to pos 7 row 2
          lda #$80+$47
          jsr sendcom
             ; twisted voting and
             ; displaying !
          jsr display
          jsr voter
             ; check whether voting failed
          beq failure
          jmp start
failure:
          jsr failureDisplay
stuck:    jmp stuck
```

Here, the adjacent calls to *display* and *voter* have been twisted – not an unlikely design flaw especially in the age of "cut-and-paste" programming. The change in the program is minimal, hardly recognizable. Though, the resulting ft-behavioural type is changed, putting emphasis on the altered structure:



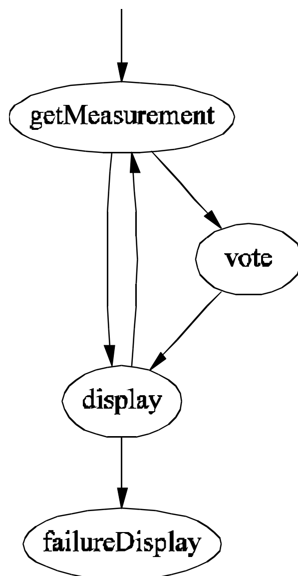
The ft-behavioural type shows that our fault-tolerance assertion is violated by the path *getMeasurement* ? *display*. SMV's shows the error by the output

```
-- specification !E(cfg != vote U cfg =
display) is false
-- as demonstrated by the following
execution sequence
state 1.1:
cfg = getMeasurement

state 1.2:
cfg = display
```

Another erroneous variant of the main routine contains a jump that was forgotten to be removed after testing:

```
start:
          jsr getMeasurement
          bne finish      ; erroneous jump
          sta measurementResult
          jsr timesTwo1
          sta voteValueA
          lda measurementResult
          jsr timesTwo2
          sta voteValueB
          lda measurementResult
          jsr timesTwo3
          sta voteValueC
             ; set cursot to pos 7 row 2
          lda #$80+$47
          jsr sendcom
          jsr voter
finish:
          jsr display
             ;check whether voting failed
          beq failure
          jmp start
failure:
          jsr failureDisplay
stuck:    jmp stuck
```

Looking at the ft-behavioural type, we see the bypass path *getMeasurement* ? *display* introduced by the erroneous branch operation:



SMV thus again delivers the counter example to denote the flaw:

```
-- specification !E(cfg != vote U cfg =
display) is false
-- as demonstrated by the following
execution sequence
state 1.1:
cfg = getMeasurement

state 1.2:
cfg = display
```

We have found our two faults here at virtually no processing costs using an automatized procedure. It is also interesting to observe that, due to the abstraction conveyed with using ft-behavioural types, we result in exactly the same counterexample for entirely different program faults.

6 Outlook

Ft-behavioural types can be the source of further analysis. As already mentioned above, they represent the abstracted structure of fault tolerance and thus can be used to categorize the fault-tolerance type of a system by structure. This can be done in several ways. One possibility, as elaborated above, is the categorization based on fulfilment of certain assertions. However, simple assertions or path descriptions may be an insufficient or cumbersome means of architecture specification.

A more sophisticated approach is the direct classification of an ft-behavioural type according to a set of reference structures and their nesting. This means employing expensive graph matching techniques but gives the benefit of gaining detailed and ordered information about a system's fault tolerance structure. E.g., the nested architecture of section 4.2 could render the regular structure description

mvp(quicksort, shellsort, insertionsort,
    rb(quicksort, shellsort, insertionsort, print))

Another application using the extracted ft-behavioural types is the derivation of numerical parameters from the programs under analysis. E.g., if there are known failure probabilities for the elements of control flow, a total probability for the failure of a system can be calculated from the derived substructures. This method of analysis is similar to fault tree analysis (overviewed e.g. in [13]) and can be

used to calculate manifold parameters. Determined by the selected vertex type subset used for abstraction, calculation can be very fine-grained, e.g. only for selected paths to determine critical paths and the like.

7 Conclusion

Analysis of software following the presented concepts promises not only a posteriori evaluation of existing software, but also a means of avoidance of design- and implementation errors. As the analysis can be done mostly automatically, the effort of checking is small for the implementer or re-engineer and can provide assurance that certain principles of fault tolerance are not undermined by implementation or design faults. This provides paths to introduce on-the-fly automatted verification components into design flows as well as to the judegement and classification of legacy software.

References

[1] A. Avizienis, The N-Version approach to fault tolerance, IEEE Trans. On Software Eng. SE-11, pp. 1491-1501, 1985.

[2] A. Avizienis, L. Chen,: On the implementation of N-Version Programming for software fault tolerance during program execution, Proc. COMPSAC'97 1997, pp. 149-155, 1997.

[3] F. Bastani, B. Cukic, V. Hilford, A. Jamoussi: Towards dependable safety-critical software, Proc WORDS'96, 1996.

[4] S.S. Brilliant, J.C. Knight, N. Leveson: The consistent comparison problem in N-Version software, ACM Software Engin. Notes pp. 29-34, 1987.

[5] E. M. Clarke, O. Grumberg, D. A. Peled: Model Checking, MIT-Press, 1999.

[6] M. Dal Cin: Zur explizit fehlertoleranten Programmierung von Parallelrechnern, in Proc. PARS Workshop, Gesellschaft für Informatik, pp. 47-55, München, April 1989.

[7] S. Gossens: Enhancing Validation with Behavioural Types, in Proc. High-Assurance System Engineering Symposium HASE 2002. IEEE, S. 201-208, Tokio, Japan, October 2002.

[8] J.C. Knight, N. Leveson: An experimental evaluation of the assumption of independence in multiversion programming, IEEE Trans Software Engin SE-12, pp. 96-109, 1986.

[9] J.-C. Laprie, J. Arlat, C. Beounes, K. Kanoun: Definition and analysis of hardware- and software fault-tolerant architectures, Computer, Vol. 23, pp. 39-51, 1990.

[10] P.A. Lee and T. Anderson, Fault Tolerance, Piniples and Practice, Springer Verlag New York, 1990.

[11] B. Prahami: Design of reliable software via general combination of N-Version Programming and Acceptance Testing, Proc. 7th Intern. Symposium on Software Reliability Engineering ISSRE'96, 1996.

[12] B. Randell, System structure for software fault tolerance. IEEE Trans. Soft. Eng., SE-1, 1975.

[13] W. G. Schneeweiss: The Fault-Tree Method, LiLoLe-Verlag, Hagen, 1999.

[14] T. Tsai: Fault tolerance via N-Modular software redundancy, Proc 28th Ann. Intern. Symp. On Fault-Tolerant Computing FTCS-28, pp 206-210, 1998.

[15] U. Voges: Software Diversität und ihr Beitrag zur Sicherheit, Proc. Entwicklung von Software-Systemen in ADA, pp. 13-27, Bremen, 1998.

[16] K. L. McMillan: The SMV System. Technical Report CS-92-131, Carnegie-Mellon-University, 1992.