Proc. 15th IEEE Intl. Symp. on Software Reliability Engineering, pp. 257-268, St. Malo, 11/04

Coverage Metrics for Continuous Function Charts

Vadim Alyokhin, Institut für Informatik, TU München, Germany
Benedikte Elbel[*] and Martin Rothfelder, Siemens AG, München, Germany
Alexander Pretschner[*], Departement Informatik, ETH Zürich, Switzerland

Abstract

Continuous Function Charts are a diagrammatical language for the specification of mixed discrete-continuous embedded systems, similar to the languages of Matlab/Simulink, and often used in the domain of transportation systems. Both control and data flows are explicitly specified when atomic units of computation are composed. The obvious way to assess the quality of integration test suites is to compute known coverage metrics for the generated code. This production code does not exhibit those structures that would make it amenable to "relevant" coverage measurements. We define a translation scheme that results in structures relevant for such measurements, apply coverage criteria for both control and data flows at the level of composition of atomic computational units, and argue for their usefulness on the grounds of detected errors.

Keywords. *Integration testing, model-based testing, mixed continuous-discrete and real-time systems, MC/DC, data flow testing, block diagrams.*

1. Introduction

Continuous Function Charts (CFCs) are a powerful graphical modeling language mainly used for engineering embedded controller devices in the domain of industrial automation and railway transportation systems. Both domains may be considered as safety-critical, and standards like EN50128 [EN01] recommend different testing strategies for systems at various levels of criticality, one of which includes structural tests. Similar to block diagrams, but more complex because of control flow issues, CFCs connect so-called function blocks that perform well-defined computations.

Functional tests are derived from the specification documents. As in other domains, it is difficult to say when one is likely to have applied a sufficient number of tests, especially if completeness of the specification is itself in question. Structural coverage criteria are used to give respective hints to testers and certification authorities. They are usually defined at the level of units, e.g., functions in C. In the domain of transportation systems, atomic CFC blocks—units—are certified independently of a particular application. When complex systems are assembled, engineers need some information w.r.t. the quality of their tests for the *integrated* (sub) system. (When we talk about quality, we refer to coverage, and not to whether or not a test case is indeed a good one.) Because certification at the unit level has already taken place, there is a need for criteria at the integration level.

It turns out that the production code itself is not particularly suited for assessing the quality of test suites w.r.t. coverage criteria. In this paper, we advocate the use of different strategies for code generation from graphical models, and show how this idea helps with testing real-world systems.

Control Flow. A natural strategy for code generators from CFCs is to have a dispatch function that calls the functions representing atomic blocks. For existing code generators, this is indeed the case: when different timing events occur, the associated computational blocks are activated. Now, the control flow of the resulting code associated with the timing class is rather linear. As a consequence, direct application of control flow based criteria to the code generated for one timing event is trivial. Since condition blocks in CFCs are all atomic, the control flow is also trivial w.r.t. criteria that take into account the structure of decisions.

We show how to identify the control flow relevant for the *integrated* functionality within the CFC. In addition, we show how to assemble atomic condition blocks into complex conditions that are amenable to the application of coverage criteria such as the modified condition/decision coverage (MC/DC) criterion. This allows the use of well-accepted coverage criteria that are based on control flow and condition coverage at the composition level of CFCs.

Data Flow. Since the structural elements of CFCs focus on the description of data flow between the functional computing blocks, data flow criteria

(*) Corresponding author: A. Pretschner, pretscha@inf.ethz.ch. Benedikte Elbel was with Siemens AG till February 2004.

[LasKor83, RapWey85] seem to be particularly suited for measuring the completeness of tests at the CFC integration level. We show how to naturally interpret the fundamental entities of data flow, definition and usage of variables, at the level of CFCs, and how to derive respective coverage metrics. Since data flow in CFCs is bound to structural elements referred to as connectors, we use these elements to introduce further coverage criteria requiring coverage at the level of connectors.

Methodology. When we defined the metrics, we did so in a top-down manner. That is to say, we analyzed existing coverage metrics and lifted them to the structural elements of CFCs, without taking into account particular examples. In order to assess our own work, we then continued by applying the new criteria to existing industrial CFCs and the respective test suites. Our metrics were able to reveal situations that had not been tested before.

Models. It is true that systems specified with CFCs exhibit an abstraction level that is rather close to the implementation. Like the graphical languages of the Matlab toolset, CFCs may be seen as a full-fledged domain-specific programming language. Models written in these languages can directly be translated into code. When compared to general-purpose languages, these domain-specific languages are, among other things, characterized by the fact that (a) they offer language constructs for domain-specific entities, and that (b) they only allow restricted operations on the syntactic elements. For instance, the respective development tools ensure that all input ports of a component are connected to the environment or some other component. These restrictions allow one to come up with tailored coverage criteria. We concentrate on what we consider the “essence” of a CFC, and assess test suites w.r.t. this essence.

Problem. Roughly, we address the problem of assessing functional tests for programs written in domain-specific non-textual languages where generated production code is unsuitable for respective coverage measurements.

Contribution. To the best of our knowledge, the paper is the first (a) to define *coverage metrics for CFCs* both for data and control flows, (b) to investigate *integration tests for mixed continuous-discrete modeling languages* on the grounds of data flow, and (c) to explicitly *synthesize complex conditions* in order to apply criteria like MC/DC.
Previous work on coverage metrics for models focuses on control flow, and on unit tests (Sec. 2). We do think that many results of this work carry over to Matlab Simulink/Stateflow diagrams, but consider this future work. We do not know of any work that applies data flow criteria to languages such as block diagrams (which becomes even more interesting in conjunction with Stateflow).

Organization. Sec. 2 lists relevant work in the area. Sec. 3 informally describes syntax and semantics of CFCs. In Sec. 4, we review traditional coverage metrics for control and data flows at the level of code, and lift them to CFCs. An algorithm is presented that identifies maximal subgraphs of a CFC exclusively consisting of logical operators. This is done to synthesize composite conditions, in order to subsequently apply respective coverage metrics. We also sketch coverage criteria for connectors, i.e., I/O ports. Sec. 5 contains experimental results. Sec. 6 discusses the issue of code instrumentation to the end of measuring coverage, and Sec. 7 concludes.

2. Related Work

Classical surveys on control and data flow criteria for code have been given in the Eighties [Nta88, FraWey88]. Today, the corresponding criteria are exposed in almost every text book on testing.
In terms of models, coverage at the level of finite state machines has been studied extensively. Among many others, Ural provides an overview [Ura92]. Lifting MC/DC to the level of extended finite state machines with a functional programming language for actions has been studied by one of the authors [Pre03]. The focus there is on test case generation. Several authors have studied the use of model checkers for the generation of test cases that satisfy a given structural criterion [RayHei01, HLS+03]. Their focus is the translation of reachability statements into temporal logics as input to a model checker.
In terms of coverage for models of continuous and mixed discrete-continuous systems, Baresel et al. have provided a recent overview [BCS+03]. In particular, they investigate the relationship between different notions of (control flow) coverage for models and for code at the level of units; integration testing and data flow are not treated. The Matlab/Simulink Model Coverage Tool concentrates on control flow and does not consider data flow and the synthesis of complex conditions.

3. Continuous Function Charts

After a brief introduction to the application domain, we use this section to provide a rough survey of both graphical syntax and computation model of CFCs.

CFCs are used for software design and implementation in the field of industrial automation. They were developed for generating automation solutions for the programmable logic controllers Siemens SIMATIC S7® and SIMATIC WinAC®. CFCs are frequently applied for developing solutions in the domain of transportation systems, building technology, and industrial process support. A development environment for design, code generation, test and simulation with CFCs is commercially available.

Process. With requirements specifications at hand, engineers build CFCs on the grounds of atomic blocks from certified libraries. Production code is subsequently generated from these CFCs. Tests derived from the requirements documents are applied to this code. The results of our study led to an intermediary step: the CFC is used for the generation of a different (inefficient) code to which the same tests are applied. Coverage is measured, and new tests are designed when coverage is not sufficient. These tests are later also applied to the production code.

3.1 Syntax

The main structural elements of CFCs are function blocks, connections between function blocks, and event classes.

Function blocks. Function blocks are the smallest units in which computational functions are described. They contain any description from simple logic functions to complex control algorithms. The function blocks may be defined in Assembler or C and are organized in certified libraries.
Blocks have input and output connectors by which parameters are passed. In order to control computational evaluation of the blocks within a CFC, timing information for a scheduling mechanism must be assigned to each block. Scheduling is driven by timing events. This is a major difference between block diagrams and CFCs.
Figure 1 contains a prototypical function block. The numbers indicate the following structural information: 1 - name of the block; 2 – input connectors; 3 – output connectors; 4 and 5 – names of the input and output connectors; 6 and 7 – type of the input and output connectors; 8 – sequence number within an event class; 9 – event class for block evaluation.

Connectors. Connectors define the interfaces of a block. Connectors are typed. Input and output connectors for a block are distinguished.

Event classes. For computation, timing information for scheduling is assigned to each block. An event class is specified, and a sequence number must be defined. It is unique within each class, defining the ordering of block computation. In Figure 1, the block belongs to class T5 with sequence number 10.

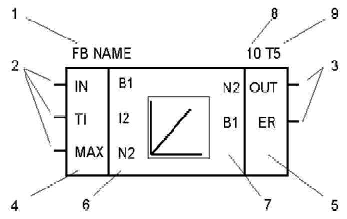


Figure 1: CFC function block

Construction of CFCs. When developing a CFC, a programmer selects function blocks from a library, places them on a worksheet, and assigns characteristics such as block name, event class and sequence number. Subsequently, the output and input connectors between the blocks are attached. Figs. 6 and 7 at the end of this paper show example CFCs. Several CFCs may be composed to a functional package. In a package, connectors are visible in all plans. Thus, blocks from different plans may be interconnected. The connection to external signals such as signals from other functional packages or hardware signals is defined at this level of construction.
CFCs may be composed hierarchically. To this end, CFC plans may be referenced as a function block in another CFC. Thus, different integration levels can be described.

3.2 Computation model

At runtime, scheduling is realized by a timing mechanism with the underlying concept of event classes. Computation is driven by timing events and interrupts. Computation of blocks within an event class is performed in the order defined by the sequence number.
When an event is raised, the assigned computational blocks are sequentially evaluated in the predefined order: values at input connectors are read, the defined computation is performed, the calculated results are assigned to the output connectors, and they are immediately available to the connected blocks.

Code. Simplifying matters, the generated production code exhibits the structure given in Figure 2. Events `e1..eQ` are raised outside this fragment. Within each event class, computations are performed for each function block $\texttt{cmp}_j$. Output vectors $\underline{\texttt{o}}_j$ are computed as a function of input vectors $\underline{\texttt{i}}_j$, and output is implicitly transferred to input connectors by address aliasing: &o$_j$ is also the address of the input connector the function block is connected to.

```
switch(event) {
  case e1:
    cmp_1(i_1,&o_1);
    cmp_2(i_2,&o_2);
    ..
    cmp_p(i_p,&o_p);
    break;
  case e2:
    cmp_p+1(i_p+1,&o_p+1);
    cmp_p+2(i_p+2,&o_p+2);
    ..
    cmp_p+q(i_p+q,&o_p+q);
    break;
   ..
  case eQ:
    cmp_p+q+1(i_p+q+1,&o_p+q+1);
    cmp_p+q+2(i_p+q+2,&o_p+q+2);
    ..
    cmp_p+q+r(i_p+q+r,&o_p+q+r);
    break;
}
```

Figure 2: Production code skeleton

In fact, except for conversions and arithmetic operations, all calls to the cmp$_j$ functions are textually replaced by the corresponding code. This is motivated by efficiency considerations in real-time domains: the length of the code does not matter as much as its speed.

Quality of tests. The production code basically consists of Q segments, one for each event class. The following three observations can be made. One, the application of *data flow criteria* corresponds to testing the interconnections between different function blocks within a CFC. We are not so much interested in data flow *within one function block* but rather at the level of composed function blocks. The above fragment does not contain explicit assignments, and the many expanded cmp$_j$ blocks convey "too much" information. Two, the application of *control flow criteria* is unlikely to convey interesting information. This is because there are no loops nor jumps, and because the involved conditions are atomic and hence trivial. Three, many expanded cmp$_j$ blocks are library functions that have been certified independently before.

4. Coverage Metrics for CFCs

This section describes the main conceptual results of this paper. An application and empirical evidence is provided in Sec. 5. After motivating a modification of the production code generator in Sec. 4.1, we briefly review traditional coverage metrics at the level of code in Sec. 4.2. We then proceed by lifting these metrics to the level of CFCs for both control (Sec. 4.3) and data flows (Sec. 4.4). Criteria for special connectors are provided in Sec. 4.5.

4.1 Code for measurements

We have seen that the generated production code is not suited for the application of coverage criteria. In terms of data flow, there are too many def-use pairs at the level of expanded cmp$_j$ blocks. Control flow is linear, and there are no or few composite conditions. For instance, a block that implements the logical AND is translated into

```
        OUT = IN1 && IN2;
```

as far as production code is concerned. While less efficient, the statement

```
        if(IN1&&IN2) OUT=1; else OUT=0;
```

is functionally equivalent but provides two branches and two definitions of OUT instead of one.

Efficiency and certification issues prevent us from simply using a production code generator that is better suited for coverage measurements. The basic idea is hence to generate code that exhibits the same "functional" behavior and that does not necessarily ensure the real-time deadlines, but is amenable to measuring the quality of test suites.

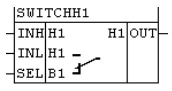
Basically, our approach amounts to modifying the code generator by (1) replacing certain code fragments by actual calls to cmp$_j$. This is done for function blocks that, in terms of coverage measurements, are deemed irrelevant at the level of composition. Since we are interested in applying traditional coverage criteria, we (2) need to gain access to the fundamental structural entities when coverage is measured.

For *control flow*, these fundamental entities are conditionals and jumps. Slightly simplifying matters, no (equivalent to) jumps exist in CFCs. Thus, for a given timing class, the control flow is rather linear. The control flow criteria are hence trivial to achieve. We hence take additional information from atomic blocks into account. For instance, a conditional block basically represents two different paths, and these two paths can be directly incorporated into the analysis. More complex coverage criteria like MC/DC rely on complex conditions. Since conditions are stated as atomic blocks within CFCs, a synthesis procedure for more complex conditions must be applied before using these criteria. Our approach to synthesizing conditions is presented in Sec. 4.3.

For *data flow*, on the other hand, the fundamental entities are writing (defining) and reading (using) accesses to variables. Variables as such do not exist in CFCs. They do exist at the level of production code. However, we are interested in connections between function blocks only—not in auxiliary variables. The output of a function block can naturally be interpreted as the definition of a "communication" variable. Input to a function block

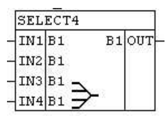
boils down to its use. Note that definitions and uses may take place at different branches of a block.
To summarize, we will generate code that contains (1) the “main” branches of the integrated structure, (2) definitions and uses of variables to the end of data flow-based testing, and (3) complex conditions to make complex control flow-based criteria applicable. This motivates the following generation procedure. For the sake of simplicity, we only sketch it. Each single function block was translated manually. Code generation for composed systems relies on this code.

Function blocks. We show how to generate code for a few chosen blocks. These are the SWITCH, SELECT, and MULT blocks. They implement simple conditionals, selections, and multiplication of two numbers. What happens is the assignment of values to output connectors as a function of the input connectors.



1. Conditionals. Depending on the value of a connector SEL, the SWITCH block decides whether the value of the INH or of the INL connector is copied to the output connector OUT. This implements a decision in CFCs; production code contains a macro at the respective places. The block is translated into a statement with two branches,

```
        if (SEL==1)
                OUT = INH
        else    OUT = INL.
```



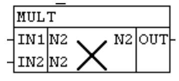
2. Selection. The SELECT block outputs 1 if exactly one of the input signals is 1. It is translated into

```
        if ((IN1 == 1) and (IN2 == 0) and
                (IN3 == 0) and (IN4 == 0))
                OUT = 1
        else if ((IN1 == 0) and (IN2 == 1)and
                (IN3 == 0) and (IN4 == 0))
                OUT = 1
        else if ((IN1 == 0) and (IN2 == 0)and
                (IN3 == 1) and (IN4 == 0))
                OUT = 1
        else if ((IN1 == 0) and (IN2 == 0)and
                (IN3 == 0) and (IN4 == 1))
                OUT = 1
        else   OUT = 0
```

while the production code is

```
        if ((IN1 & 1) + (IN2 & 1) + (IN3 & 1)
                 + (IN4 & 1))
            OUT = 1
        else OUT = 0
```

with fewer branches, fewer definitions, and no composite conditions (& is the bitwise AND).



3. Multiplication. The MULT block implements multiplication of two 2-byte integers provided at connectors IN1 and IN2. The result is made accessible at connector OUT. The arithmetic operation involves no decisions and is described as function

```
        OUT = compute_MULT_OUT (IN1, IN2)
```

which refers to an external function that performs multiplication. The actual code that includes checks is hidden in this function.
This specialized generation of code *at the level of single function blocks* is only seemingly in contradiction with one of our initial statements: we claimed that measuring coverage at this level is a dubious endeavor because of the certification that has already taken place. The point is that we identified those function blocks with “interesting” conditions for the integrated functionality and put them at the *top level* of the differently generated code rather than calling corresponding functions. This is, for instance, the case for the multiplication but *not* for the SWITCH and SELECT blocks. However, the code generated this way is not a mere macro expansion: instead, it contains the “relevant” parts only, and it omits “irrelevant” details.

Composition. Rather than assigning output by calls by reference, we introduce explicit assignments. This is relevant for data flow measurements only. Composite conditions for control flow measurements are treated in Sec. 4.3.

4.2 Traditional metrics

In the following, we briefly review classical coverage metrics at the level of code. Structural coverage criteria have initially been developed with the aim of selecting test data or of judging test data adequacy [GooGer75, How76]. They are defined in terms of the internal structure of a piece of software. They can thus be applied even if completeness of a specification is in question, or if no specification at all is at hand. Efficiency and the ability to detect failures of different structural coverage criteria were controversially discussed in the literature.

Control Flow. Structural coverage criteria that use control flow as a reference were introduced as control flow based criteria [How76, Mye79]. Statement coverage requires every statement to be executed at least once during testing. Branch coverage requires every branch to be executed at least once: every decision evaluates to both true and false. Requiring all program paths to be executed at least once has been introduced as a powerful yet generally unachievable criterion. Selection of special paths to be covered led to the development of coverage criteria such as LCSAJ or boundary-interior coverage [How76, WHH80].

Data Flow. Examination of the data flow in programs led to further requirements on paths to be covered during testing [LasKor83, RapWey85]. As a basic criterion, the all-defs criterion requests for all definitions of a variable a dynamic test case executing a definition free path (with respect to this variable) from its definition to one of its uses. The more demanding all-uses criterion requires a dynamic test case for covering a definition free path from each definition of the variable to all its uses. To fulfill the most demanding criterion of this family, the all-def-use-paths criterion, all such paths have to be covered. These and further criteria for covering the data flow, such as required pairs testing, were introduced and discussed as data flow criteria.

Complex Conditions. Control and data flow oriented techniques do not take into account the internal complexity of compound predicates used in the decisions of the program. Since predicates determine the execution path for a given input and thus have a high impact on the dynamic behavior of the program, additional coverage metrics have been defined with respect to the internal structure of compound predicates [Mye76, ChiMil94].

Conditions are regarded as covered if they have been evaluated to the Boolean constants true and false at least once during testing. The weakest criterion, referred to as condition coverage, requires evaluation of all atomic conditions to true and false. Condition/decision coverage has then been defined with respect to atomic and top-level conditions. Requiring the coverage of all conditions—atomic, intermediate and top-level—has been introduced as minimal multiple condition coverage. Demanding each compound predicate to be covered with all possible combinations of Boolean values of its atomic conditions led to multiple condition coverage. This criterion is costly to achieve since it requires 2^m combinations for covering a compound predicate with *m* atomic conditions. With the aim of thoroughly examining each atomic condition in a compound predicate, modified condition/decision coverage (MC/DC) has been defined. In order to satisfy this criterion, each atomic condition must influence the predicate at least once, while all other independent conditions stay unchanged. For a compound predicate with m conditions that can be varied independently, this criterion can be achieved with *m+1* combinations.

4.3 Control Flow Metrics for CFCs

In this section we show how techniques that are based on control flow can be used to evaluate completeness of a test suite for CFCs. The graphical representation of a CFC does not allow direct application of the coverage criteria. In order to convey the meaning of a metrics for a graphic plan, the metrics must be applied to the generated code. Because the application of well-known coverage criteria based on control flow is rather trivial, we focus on the synthesis of complex conditions. That is to say, in order to apply criteria that take into account the structure of complex conditions, we would like to measure coverage on a piece of code

```
        r =    (c1 AND c2) OR c3
```

rather than

```
        r1 =   c1 AND c2
        r =    r1 OR c3.
```

The semantic differences between these two expressions are not relevant in our context.

Synthesis of composed conditions. CFCs provide blocks for the logical *and*, and *or* operators (*not* is handled by means of connectors, see the example below). They are interconnected to implement complex conditions/decisions. In order to apply coverage metrics that take into account the structure of conditions, composed conditions have to be synthesized. For these complex logical conditions, coverage criteria like minimum multiple condition or MC/DC can be applied.

The idea is to identify maximum subgraphs of a CFC that consist only of logical operators, and to compute the corresponding complex condition. In the following, we assume that all function blocks belong to the same event class, and for the sake of simplicity, we consider binary conjunction and disjunction only. Let F denote an array of function blocks in *reverse order of execution* (element 8 in Figure 1). This array contains only the logical function blocks "AND" and "OR" of a complex module. Furthermore, let *pred* denote a function that takes a function block and an input connector, *i*, and returns the function block that contains the output connector that is connected to *i*. The range of pred may be a superset of F. In Figure 4, for instance, we have

```
        pred(FRMD_10, IN1) = FRMD_9.
```

In case there is no predecessor, *pred* returns the name of the input signal. The algorithm then works as follows. Function *max_subgraph* defined in Figure 3 is executed for each f∈F. This yields an array of possibly complex conditions, C:

```
        for i=1 .. |F|
                C[i] = max_subgraph(F[i]).
```

An element C[i] is the empty string if F[i] is a logical block that has become part of a more complex synthesized condition. *max_subgraph* is recursively applied to those function blocks that yield the input for connectors in1 and in2, and their result is connected by "and" or by "or", depending on the nature of f. *is_neg* returns TRUE if its argument is a negated connector (see below), and false otherwise.

in: function block f
out: string of synthesized conditions with f at the top level
side effects: `mark_visited` changes status of blocks (initially FALSE)

```
 max_subgraph(f)
   if was_visited(f) then return ""
   else
     c1 = max_subgraph(pred(f.in1))
     if is_neg(f.in1)
              then c = "(not(" + c1 + ")"
              else c = "((" + c1 + ")"
     if is_and(f)    then c += " and "
                     else c += " or "
     c2 = max_subgraph(pred(f.in2))
     if is_neg(f.in2)
              then c += "not(" + c2 + "))"
              else c += "(" + c2 + "))"
     mark_visited(f)
     return c
  end
```

Figure 3: Synthesizing complex conditions

mark_visited(f) makes sure that subsequent calls to *was_visited*(f) return TRUE instead of FALSE. We need this function to ensure maximality of the synthesized condition. We will show in Sec. 6 how these synthesized conditions are used for measuring coverage with criteria such as MC/DC.

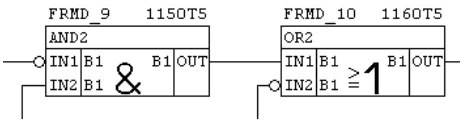


Figure 4: Composite condition

Example. Figure 4, a subgraph of Figure 7, shows two interconnected blocks, AND2 and OR2. Circles on connectors indicate negation. The following condition C[2] is derived from these two blocks:

```
[(FRMD_9.IN1 == FALSE) AND (FRMD_9.IN2 ==
     TRUE)] OR (FRMD_10.IN2 == FALSE)
```

where FRMD_9.IN1 and FRMD_9.IN2 are the values of signals at connectors IN1 and IN2 in block FRMD_9, and FRMD_10.IN2 is the value of the signal at connector IN2 in block FRMD_10.

4.4 Data Flow Metrics for CFCs

Since CFCs are essentially data flow diagrams, traditional criteria based on data flow appear to be particularly suited for use with CFCs. The basis for applying data flow criteria is the data flow graph. Roughly, that is a control flow graph enriched with data flow information describing definitions (def), predicative use (p-use) and computational use (c-use) of variables. As mentioned above, data flow criteria then require paths between definitions and uses of variables to be covered.

In CFCs, paths between definitions and uses are described with solid lines between different blocks. To achieve data flow coverage within the corresponding code, it is necessary to cover these connections. Of course, this is motivated by our initial desire to measure coverage at the level of composition. Nevertheless, a simple connector-based coverage—i.e., one that requires that a value must be written at each connector—is not sufficient. This is because within a block, due to its internal control structure, several definitions and uses may appear. In this case, one connection may well correspond to more than one def-use-pair. Figure 5 shows a part of the code (not production code) that corresponds to the CFC in Fig. 6 where blocks TEMP_ML{3,4} are located in the second rightmost column, blocks TEMP_CP{1,2} are located in the middle column, and blocks TEMP_SW{2,3} are in the rightmost column. The code contains three lines in bold face, the first two of which define a value, and the last of which uses the corresponding value. In Fig.6, the corresponding connection between two function blocks is also given in bold face. In the following, the first component of a pair denotes the value of TEMP_CP1.OUT, and the second component denotes TEMP_CP2.OUT. The test data {(1,0),(0,0)} satisfies all-defs coverage but not branch coverage while the test data {(0,0),(1,1)} satisfies branch coverage but not all-defs. While the relationship between def-use and branch coverage is well known [Nta88], we give the example in order to convey the relationship with CFCs.

```
TEMP_ML3.OUT =
  compute_MULT_OUT(TEMP_SB3.OUT, 0.333)
TEMP_ML4.OUT =
  compute_MULT_OUT(TEMP_SB1.OUT, 0.333)
if (TEMP_CP1.OUT)
   then TEMP_SW3.OUT = TEMP_ML3.OUT
   else TEMP_SW3.OUT = TEMP_ML1.OUT
TEMP_ML2.OUT =
  compute_MULT_OUT(TEMP_AD2.OUT, 0.25)
if (TEMP_CP2.OUT)
   then TEMP_SW2.OUT = TEMP_ML4.OUT
   else TEMP_SW2.OUT = TEMP_SW3.OUT
if (TEMP_CP3.OUT)
   then $62M7W00 = TEMP_ML2.OUT
   else $62M7W00 = TEMP_SW2.OUT
```

Figure 5: Part of the pseudo code for the CFC in Fig. 6

4.5 Metrics for connectors

In this section, we present some additional criteria that are specific to CFCs. These criteria are defined independently of blocks in a CFC. Instead, connectors are used as a basis. These criteria cater for the test of signals at connectors where the notions of equivalence classes, limit values, and special values make an intuitive sense. We only consider connectors that allow signals within a whole range of values. These include integer signals, binary signals, and floating point signals:

- For integer signals, the special value zero and the minimum and maximum values are selected. As an example, consider a signed 1-byte integer signal *S*. The connector-based criterion then requires test cases with the following values: $S < -128$; $S = -128$; $-128 < S < -1$; $S = -1$; $S = 0$; $S = 1$; $1 < S < 127$; $S = 127$ and $S > 127$.
- For binary connectors—whose range is {0,1}—these two values are required for the tests.
- For floating point signals, negative and positive values are required, and so is the value 0. The test cases include values that are almost zero. For 48-bits floating point signals, for instance, this is the value 0.9e-2466. In critical cases, it might also be interesting to apply a value between zero and the value: $0 < S < 0.9\text{e-}2466$.

One may argue that these criteria correspond to the evaluation of equivalence classes for internal variables. Such a criterion is indeed demanding, and may even be unachievable. Nevertheless, for failure-prevention in safety-relevant functions it may be suitable to try to provoke over- or underflow of variables and faulty internal data states by assigning boundary values to data transferred internally.

5. Application

In order to demonstrate the usefulness of the defined criteria, we now present results from their application to CFCs from real-world projects. Secs. 5.1 and 5.2 discuss the application to CFCs containing mainly arithmetical, and logical blocks, respectively. In both examples, coverage criteria were applied to test cases previously defined w.r.t. a given functional specification. We were able to identify structural elements—and, equivalently, functionality—of the CFCs that had not been covered by the given functional test cases. As a consequence, additional test cases were defined and run. The results of these additional test cases turned out to be useful for software quality assurance, as detailed below.

5.1 CFCs with arithmetics prevailing

The CFC used as a first example is presented in Figure 6. It computes the average of four temperature values. Its functional specification requires the maximum value of the four temperature values to be excluded from the computation of the average if it exceeds the average of the other values by a factor of 1.1. Analogously, if the minimum temperature value is smaller than the average value from the other three temperature values multiplied by 0.9, it has to be excluded as well. If both the minimum and maximum values are excluded from the computation, the average is calculated from the remaining two temperature values. The case of negative temperatures is handled in the actual system but omitted here for the sake of simplicity.

Table 1 shows the original functional test cases that were derived from the functional specification sketched above. The structural criteria presented above were applied after running these test cases. All three of them were necessary to achieve complete branch coverage.

Table 1: Original test cases for computation of average

	v1	v2	v3	V4	Exp.Result
Test 1	20°C	21°C	22°C	23°C	21.5°C
Test 2	20°C	21°C	22°C	25°C	21°C
Test 3	16°C	21°C	22°C	25°C	21.5°C

All-uses coverage. Since one of the def-use pairs, namely that between the SWITCH blocks TEMP_SW3 and TEMP_SW2 printed in bold in Figure 5, was not covered by the three test cases, a def-use coverage of 98% is achieved by the three test cases. Additional test data v1=16°C, v2=21°C, v3=22°C, and v4=23°C with an expected result of 22°C was designed in order to test the missing def-use-pair. This test data showed a deviation from the specification: the specification requires the value 16°C to be excluded from the computation because $16 < (21 + 22 + 23) / 3 * 0.9$. The average value of the remaining three signals is 22°C. However, the result computed by the CFC is 22.64°C. The error is caused by a wrong calling ordering of the blocks that causes a value to be erroneously taken from the previous computation step.

Connector-based coverage. The connector-based criteria presented in Sec. 4.5 require test cases provoking underflow, overflow and truncation of values at numeric connectors. For internal numeric connectors between arithmetic blocks of a CFC, such requirements may be very demanding, if not unachievable. Thus, connector based coverage must be interpreted with care and should be monitored only for critical functionality. In the example presented above, a connector-based coverage of only 3.6% was achieved by the functional test cases.

5.2 CFCs with logics prevailing

A second example for the application of coverage criteria is presented in Fig. 7. It deals with energy management for an air conditioning system in a two-cabin vehicle. Two states of the power supply are to be differentiated: normal operation with four air conditioning systems takes place in the state "full power supply". In state "reduced power supply", only one air conditioning system of four may be used. Thus, if several air conditioning systems are switched on at the same time in this state, only one may begin to operate. If one of the air conditioning systems is running in state "reduced power supply" and a second one is switched on, then the first air conditioning system must be switched off.

The CFC describes the switching logic in one of the cabins. Signal $61Z5002 reports whether an air conditioning system in the other cabin is switched on. Signals $61E5903 to $61E5906 encode the state of the four air conditioning systems in the cabin. Signal $31M5000 indicates reduction of power supply. B/H and H/B blocks perform a binary to hexadecimal translation and vice-versa. The lower part of the CFC implements the behavior where the „reduced power supply“ signal makes sure that an AC is switched off when a new one is switched on. The upper part decides whether all ACs should be switched off. The six test cases in Table 2 had been derived from the functional specification.

Table 2: Original test cases for the AC system

Test 1	- Full power supply - All AC systems switched off - Expected Result: All ACs off.
Test 2	- Full power supply - All ACs switched on - Expected Result: All ACs working
Test 3	- Reduced power supply - No AC working in cabin B - All ACs in cabin A are switched on - Expected Result: All ACs switched off
Test 4	- Reduced power supply - All ACs in both cabins are off - AC #1 in cabin A is switched on - Expected Result: AC #1 working
Test 5	- Reduced power supply; - AC #1 in cabin A working. All other ACs in both cabins are switched off - AC #2 in cabin A is switched on - Expected Result: AC #1 is switched off, AC #2 is switched on
Test 6	- Reduced power supply; - One AC is switched on in cabin A; all other ACs are off. - AC #2 is switched on in cabin B - Expected Result: All ACs switched off in cabin A

Branch coverage. The functional test cases covered 91% of all branches. For full branch coverage, additional test cases had to be designed. Roughly speaking, these switched on and off all types of air conditioning systems in the state "reduced power supply".

MC/DC coverage. In the CFC in Fig. 7, one complex logical condition could be sythesized. The condition consists of the two blocks `FRMD_9` (and) and `FRMD_10` (or) with input signals `$61Z5002`, `FRMD_8.OUT` and `KLM_FRSP.OUT`. The synthesized condition can be described as follows (cf. Sec. 4.3):

```
[not($61Z5002) AND (FRMD_8.OUT)] OR
          not(KLM_FRSP.OUT)
```

The combinations of truth values tested by the functional test cases for the complex condition are described in rows 1-6 of Table 3. Test cases 1 and 2 as well as 4 and 5 are identical only w.r.t. the given signals. They do not cover the signal $61Z5002 w.r.t. MC/DC. For achieving complete MC/DC coverage, the additional test case 7 had to be designed, covering signal $61Z5002 in combination with test case 4.

Table 3: Projection of tests in Table 2

Test	NOT $61Z5002	frmd_8. out	NOT klm_frsp. out	Result
1	T	F	T	T
2	T	F	T	T
3	T	F	F	F
4	T	T	F	T
5	T	T	F	T
6	F	F	F	F
7	F	T	F	F

The additional test case 7 helped to discover a non-specified behavior. This error leads to a malfunction occurring in state "reduced power supply", with one air conditioning system running in one of the cabins: in this state, if one tries to switch on a second air conditioning system in the other cabin, the new air conditioning cannot be activated. Nevertheless, a flag is set that the new air conditioning system is switched on, requiring to stop the air conditioning system previously at work. Thus, all air conditioning systems are deactivated, while the internal data state erroneously indicates the first one to be active.

This is an argument in favor of using MC/DC, and this is consistent with the findings of Dupuy and Leveson [DupLev00].

All-uses coverage. The six functional test cases achieved an all-uses coverage of 82%. As with branch covererage, full coverage could simply be achieved by defining further test cases requiring all types of air conditioning systems to be switched on and of during testing in the state “reduced power supply”.

Connector-based coverage. From all connector-based criteria, only criteria for binary connectors are applicable for the CFC plan. Full coverage could be achieved with five out of the six test cases. This demonstrates that criteria for binary connectors may be regarded as a weak coverage criterion that can easily be achieved.

6. Instrumentation of Code

We have seen that different strategies of code generation impact coverage measurements. These latter necessitate the instrumentation of code, which can be done by external tools. One can also directly encode the measurements into the code. While this means re-inventing the wheel to a large extent, its benefit is a higher degree of flexibility.

Recall that the production code roughly looks as given in Figure 2. Modifying the code generator yields code that (1) expands some function calls, (2) replaces some fragments by calls, and (3) contains explicit “communication variables”. Further adjustments are necessary for coverage measurements.

- For measurement purposes, copies of all i_j and o_k variables (Figure 2) are introduced, i_j' and o_j'.
- Each `cmp` statement is preceded by a set of copying statements $i_j'=i_j$. This makes sure that the dummy code subsequently works on adequate values.
- Some (Sec. 4.1) of the calls to `cmp` are expanded into actual code that works on these redundant variables. This may go beyond simple macro expansion of the originial production code. For instance, the example of the `AND` block shows that additional branches are introduced. Because this code works on the redundant variables, there is no interference with the original functionality.
- The calls to `cmp` that have not been expanded, `cmp(`$\underline{i}$`,&`$\underline{o}$`);` are followed by a set of copy statements $o_j'=o_j$ for $1 \le j \le M$ where M is the length of vector $\underline{o}$. This makes sure that the correct values are used subsequently.
- Synthesized composite conditions (Sec. 4.3) are inserted before the first corresponding `cmp` function is executed. The `then` and `else` branches both lead to empty statements, i.e., they do not modify any values.

The motivation for introducing shadow variables, i' and o', is that they allow us to use simplified expanded versions of some function blocks. By doing so, we can ignore all the internal checks in a function block when it comes to coverage measurements.

This procedure inserts new branches and conditions into the code. The relationship between coverage on the original and on this altered code is that whenever coverage on the latter increases, it also does so on the former. However, recall that we are not so much interested in actual code coverage—just consider the many checks that defensive programming introduces—but rather in coverage of the main functionality.

7. Conclusions

Summary. This paper provides a solution to the problem of assessing test suites for CFCs at the level of composition. Production code at the level of composition does not exhibit the structures that would make it amenable to coverage measurements with results that can be interpreted. We have presented a way of generating code that is too inefficient to serve as production code. On the other hand, this code does allow one to apply coverage criteria at the level of composed function blocks. The straightforward interpretation of connections as variables gives rise to def-use pairs, and the synthesis of complex conditions makes this code amenable to criteria such as MC/DC. Coverage is measured for the “relevant” parts of CFCs only. We are confident that our results carry over to the similar languages of Matlab/Simulink where coverage measurements at the level of single models are good practice, but not at the integration level of components.

Assessment. Our experiments show the usefulness of the criteria for they revealed previously untested situations in commercial applications. They are currently used at Siemens for the assessment of test suites. We do not use coverage criteria for the generation of test cases.

We are aware that there is no conclusive evidence on the “quality” of coverage metrics w.r.t. their power of revealing failures, and that a controversial discussion on the benefits of random testing vs. structural testing persists [HamTay90, Nta98, Gut99]. However, because our experiments revealed previously untested situations, and because coverage metrics provide some number that can be interpreted as progress, it was decided to incorporate the use of the coverage metrics of this paper into the development process.

Future Work. In terms of ongoing empirical investigations, we try to get further evidence at whether or not the use of coverage metrics pays off in our particular application domain. This is regardless of the fact that coverage measurements are recommended by the certification authorities. As far as technology is concerned, we are manually translating more and more function blocks that somehow involve conditions, and whose computation is not dispatched to external functions but is rather performed at the top level of the generated code (Sec. 4.1). Furthermore, we consider it interesting to measure coverage w.r.t. other metrics and to compare the ability to detect failures in our domain of graphical implementations of railroad transportation systems.

Because the languages of Matlab Simulink/Stateflow are similar to CFCs, we consider an adaptation to these languages and other application domains, such as automotive systems, to be straightforward.

Regardless of whether or not one accepts CFCs as a language for model-based development, the work in this paper is an instance of a more general issue, namely how to use coverage criteria in the context of model-based development. Dedicated criteria can be defined, or classical criteria can be applied to generated code. (Inconclusive) past studies on the relationship between implementation coverage and error detection must be complemented by further studies that take into account the role of models [WGS94].

Finally, we did not treat the issue of automatically generating test cases in this article. The rather simple execution semantics of CFCs is similar to that of the CASE tool AutoFocus [HSE97] for which an industrially relevant body of automatic test case generation technology has been developed [PSA+04, Pre03]. Consequently, it appears feasible to not only *assess* the quality of test suites but instead automatically *generate* test suites that satisfy a given coverage criterion by construction

References

[BCS+03] Baresel, A., Conrad, M., Sadeghipour, S., Wegener, J.: *The Interplay between Model Coverage and Code Coverage*, Proc. EuroCAST, 2003.

[ChiMil94] Chilenski, J., Miller, S.: *Applicability of modified condition/decision coverage to software testing*, Software Engineering Journal, pages 193—200, September 1994.

[DupLev00] Dupuy, A., Leveson, N.: *An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software.* Proc. Digital Aviations Systems Conference, October 2000

[EN01] European Committee for Electrotechnical Standardization: *EN 50128:2001—Railway applications—Communications, signaling and processing systems--Software for Railway Control and Protection Systems*, 2001.

[FraWey88] Frankl, P., Weyuker, E.: *An applicable family of data-flow testing criteria.* IEEE TSE 14(10):1483-1498, 1988.

[GooGer75] Goodenough, J., Gerhart, S.: *Toward a theory of test data selection*, IEEE TSE 1(2):156—173, 1975.

[Gut98] Gutjahr, W.: *Partition Testing vs. Random Testing: The Influence of Uncertainty.* IEEE TSE 25(5): 661-674, 1999.

[HamTay90] Hamlet, D., Taylor, R.: *Partition Testing does not inspire confidence.* IEEE TSE 16(12):1402-1411, 1990.

[HLS+03] Hong, H., Lee, I., Sokolsky, O., Ural, H.: *A Temporal Logic Based Theory of Test Coverage and Generation*, Proc. TACAS'02, pp. 327-341.

[How76] Howden, W.: *Reliability of the Path Testing Strategy*, IEEE TSE 2(3): 208-215, 1976.

[HSE97] Huber, F., Schätz, B., Einert, G.: *Consistent Graphical Specification of Distributed Systems*, Proc. FME'91, pp. 122-141, 1997.

[LasKor83] Laski, J., Korel, B.: *A data flow oriented program testing strategy*, IEEE TSE 9(3):347—354, 1983.

[Mye79] Myers, G.: *The Art of Software Testing*, Wiley, New York, 1979.

[Nta88] Ntafos, S.: *A comparison of some structural testing strategies*, IEEE TSE 14(6):868-874, 1988.

[Nta98] Ntafos, S.: *On random and partition testing.* Proc. Intl. symp. on Software Testing and Analysis, pp. 42—48, 1998.

[Pre03] Pretschner, A.: *Compositional Generation of MC/DC Test Suites*, Electronic Notes in Theoretical Computer Science 82(6):1-11,2003.

[PSA+04] Pretschner, A., Slotosch, O., Aiglstorfer, E., Kriebel, S.: *Model-Based Test Case Generation for Real: The Inhouse Card Case Study*, Software Tools for Technology Transfer 5(2-3):140—157, 2004.

[RapWey85] Rapps S., Weyuker, E.: *Selecting Software Test Data Using Data Flow Information*, IEEE TSE 11(4):367-375, April 1985.

[RayHei01] Rayadurgan, S., Heimdahl, M.: *Coverage Based Test Case Generation Using Model Checkers.* Proc. 8th Intl. Conf. and Workshop on the Eng. of Computer-Based Systems, pp. 83-93, 2001.

[Ura92] Ural, H.: *Formal Methods for Test Sequence Generation*, Computer Communications 15(5):311-3254, 1992.

[WHH80] Woodward, M., Hedley, D., Hennell, M.: *Experience with path analysis and testing of programs*, IEEE TSE 6(3): 278—286, September 1980.

[WGS94] Weyuker, E., Goradia, T., Singh, A.: *Automatically Generating Test Data from a Boolean Specification*. IEEE TSE 20(5):353-363, May 1994.

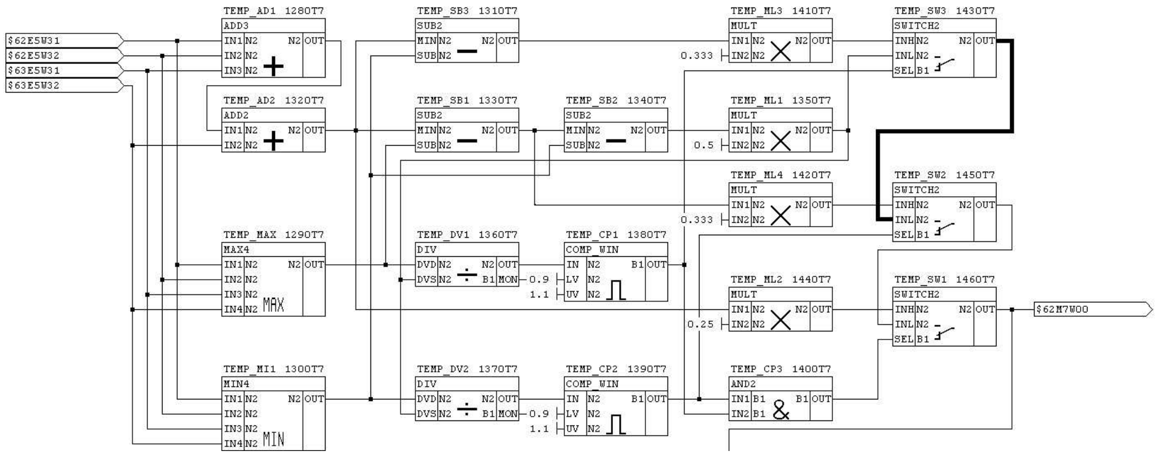


Figure 6: Example of a CFC mainly consisting of arithmetic blocks

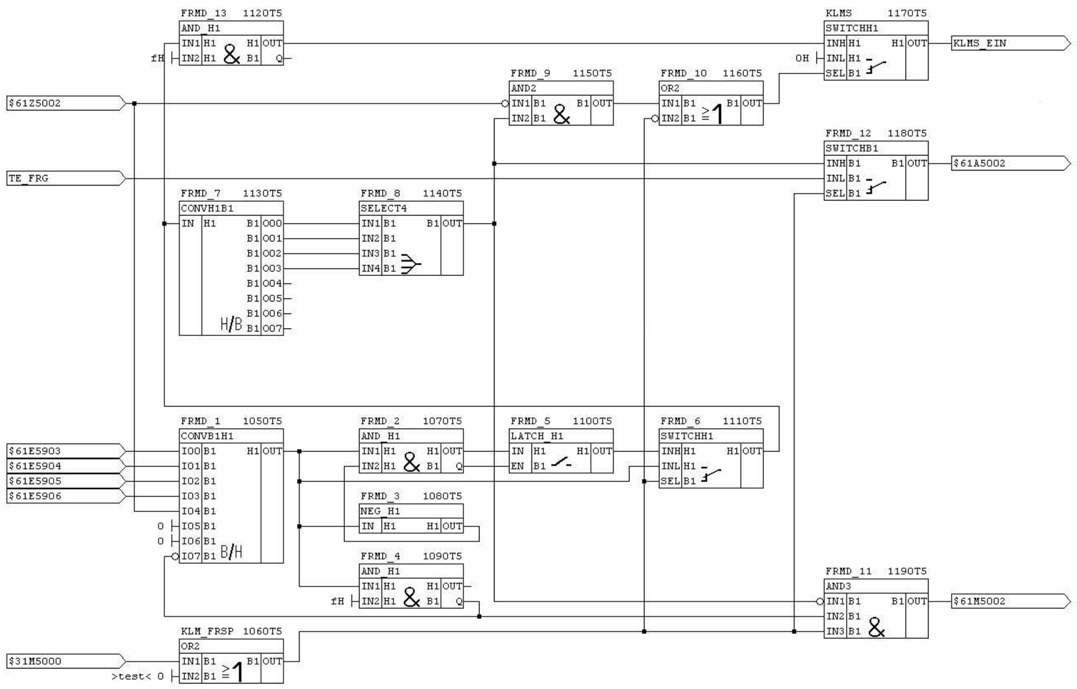


Figure 7: Example of a CFC mainly consisting of logical blocks