# Slope Testing for Activity Diagrams and Safety Critical Software

Andreas Holzer Visar Januzaj Stefan Kugele Christian Schallhart
Michael Tautschnig Helmut Veith Boris Langer

Formal Methods in Systems Engineering, FB Informatik, TU Darmstadt

holzer@forsyte.de, januzaj@forsyte.de, kugele@forsyte.de,
schallhart@forsyte.de, tautschnig@forsyte.de, veith@forsyte.de,
boris.langer@diehl-aerospace.de

Technical Report TUD-CS-2009-0184

# Slope Testing for Activity Diagrams and Safety Critical Software *

Andreas Holzer[1], Visar Januzaj[1], Stefan Kugele[1], Christian Schallhart[1], Michael Tautschnig[1], Helmut Veith[1], and Boris Langer[2]

[1] Formal Methods in Systems Engineering, FB Informatik, TU Darmstadt
{holzer, januzaj, kugele, schallhart, tautschnig, veith}@forsyte.de
[2] Diehl Aerospace GmbH
boris.langer@diehl-aerospace.de

**Abstract.** Formal system modeling and rigorous validation techniques have become a corner stone in the development practice for safety critical systems. It is characteristic for model-based approaches that the relationship between the model and its implementation needs to be monitored and ultimately brought to conformance. To bridge the gap between model and implementation, the current paper proposes a new methodology called *slope testing* where we concretize an abstract test suite covering the model to obtain a corresponding concrete test suite on the implementation. In this way, our method is able to systematically expose the potential deficiencies in the mapping between model and code. Motivated by the avionic certification standard DO-178B, we introduce slope testing in a prototypical process which is based upon UML activity diagrams and ANSI-C as the respective modeling and implementation languages. Our implementation makes use of the test case generator FShell which automatically generates the required test suites for activity diagrams and source code.

## 1 Introduction

As the complexity of safety critical software is increasing, industry and regulators have defined standards such as DO-178B [1] to assure software correctness. The standards are putting pressure on suppliers to obtain approved software models as early as possible, and to develop implementations whose conformance to the models is supported by strong evidence. It is crucial to avoid or at least detect potential deviances as early as possible: If the certification authorities are not convinced about the system's safety, they will request a re-development of the questionable subsystems. The high investment already made leaves the supplier usually no other choice than to implement the requested changes.

This situation is reflected in a real-life emphasis on analysis and design which we observed in the avionics industry. For Design Assurance Level (DAL) A,

* Supported by DFG grant FORTAS and BMWI grant 20H0804B in the frame of LuFo IV-2 project INTECO

which is the highest assurance level in DO-178B on a range from A to E, a typical avionics supplier company will spend 15% of time on analysis, 25% on design, 30% on implementation, 10% on requirements validation, and 20% on implementation verification. Decreasing the assurance level from A to B saves only 5% in the verification expenses while the step from B to C cuts them half. The validation costs are not affected by the aspired DAL.

Concerning verification, DO-178B is essentially asking for a test suite satisfying the following conditions:[3]

**C1** The test suite is derived from the requirements *independently of the source code.*

**C2** The test suite has to assure *requirements coverage*; both high level and low level requirements have to be considered.

**C3** On the source code, the test suite has to achieve *structural coverage.* The exact coverage notion depends on the required DAL. For instance, DAL A requires modified condition/decision coverage (MC/DC) as well as data and control coupling coverage on the source code.

**C4** The test suite has to be executed on the implementation *without errors.*

When executed on a program under test, the test suite may indicate problems by violating either **C3** or **C4**, i.e., either MC/DC is not achieved (**C3**), or an error is found (**C4**). In contrast, conditions **C1** and **C2** are process criteria which cannot be formally verified; in a tool chain, errors in these conditions only become visible through process inherent consistency checks and subsequent violations of **C3** and **C4**.

In this way, DO-178B attempts to reveal errors in process criteria **C1** and **C2** through formal criteria **C3** and **C4**.[4] The most striking difficulty here is, of course, the apparent contradiction between **C1** and **C3**. *How is it possible to achieve MC/DC in a black box manner ?*

It is witness to the genius of DO-178B that extremely careful requirements engineering is the only solution to this dilemma. In essence, **C1** and **C3** say that at the time of certification, the requirements must contain sufficient detail about the control flow of the implementation to *predict* MC/DC. Thus, the requirements should preclude "creativity" in coding.

Industrial practice thus demands a powerful mechanism to support the smooth transition from abstract requirements to very detailed models. This is the reason why UML activity diagrams [2] have become a very popular tool in avionics engineering. Originally intended for high level requirements, activity diagrams have sufficient expressive power to be a viable alternative to represent low level requirements. The practice that activity diagrams are already used to express low level requirements on a day-to-day basis at our industrial collaborators is

[3] This paper discusses key aspects of DO-178B; rather than claiming full conformance, we introduce notation and terminology for better understanding of the standards.

[4] For certification purposes, the supplier must argue—usually in an informal and textual manner—that **C1** and **C2** have been satisfied.

compatible with the expected more rigorous requirements of the successor standard DO-178C, which will require a formal modeling of low level requirements. In preparation for the following discussion we note an asymmetry in DO-178B: While low level requirements have to be specified formally, high level requirements are *required* to be understandable by a general audience, i.e., to be textual.

The main contribution of this paper is a systematic testing methodology which provides tool support for DO-178B conformant testing in avionics software and similar settings, for instance in the automotive industry. Our method aims to reflect the degree of abstraction which separates the model from the implementation, e. g., an activity diagram from its corresponding C code. If the model exhibits a high degree of abstraction, or, equivalently, if the implementation enjoys a significant amount of behavioral liberty, the conceptual slope between the models is steep, and further refinement is necessary; otherwise, the slope is flat, and conformance is achieved. To reflect the conceptual gap between the two models, we refer to our method as *"slope testing"*.

In the implementation of slope testing discussed here, we will use activity diagrams and ANSI-C programs to serve as modeling and implementation languages respectively; the results of this paper can be easily transferred to other settings. Note, however, that our tool chain makes heavy use of FShell [3–5], a configurable ("query-driven") test case generator for C source code which was developed in our group during the last years; in fact, we shall see below that the ability to generate white box test cases with high precision is essential to the methodology of this paper.

Slope testing instantiates the generic picture of DO-178B with the following steps:

**S1 Test Suite Generation.** Following industrial practice, we use activity diagrams to express low level requirements. In an automatic step, we translate activity diagrams into fragmentary C code containing control flow statements, stubs, and labels.

- **S1a** Using FShell, we compute a test suite which achieves MC/DC on this fragmentary program (or another generic coverage criterion, depending on the assurance level needed). Essentially, this aims at satisfying **C3** and **C1** simultaneously.
- **S1b** In addition, the test engineer spells out the tests needed for requirements coverage in FQL, the input language of FShell. Importantly, FQL supports the succinct and declarative description of user defined coverage criteria. Oftentimes, hundreds of concrete test cases can be described by a single FQL query, and thus it is easier to link the test cases to the original (possibly textual) requirements. This step enlarges the test suite from **S1a** and assures **C2**.

**S2 Test Suite Concretization.** In this step, each of the test cases generated in step **S1** is realized as a concrete test case on the implementation. In order to fulfill the black box criterion implicit in **C1**, we need to verify that there is a 1-1 correspondence between the test cases from the activity diagrams and the test cases on the implementation. At this point, the following two **deficiencies** in the model-implementation slope can occur:

**D1 Implementation Poverty.** If some test case cannot be concretized, then the model contains behavior not implemented in source code. The reason can be either on the side of an incomplete implementation, but also in erroneous or imprecisely formulated requirements. Note that implementation poverty can be spurious, if the model does not reflect relevant dependencies of the involved control flow decisions.

**D2 Implementation Liberty.** The implementation contains more than one concrete control flow path for some abstract test case. This situation expresses that the activity diagram is an abstraction of the implementation, and thus, the model does not fully determine the control flow of the implementation.

If neither **D1** nor **D2** occurs, then for each abstract test case there exists *exactly one* control flow path to be captured by an implementation test case which matches the abstract test case, and vice versa.

If the concretization step **S2** fails, then the resulting concrete test depends crucially on the implementation, violating condition **C1**. In addition, condition **C2** should be checked since some test cases proved to be over- or underspecified, corresponding to deficiencies **D1** and **D2**.

**S3 Test Suite Evaluation.** In this phase, the test suite obtained in **S2** is executed on the implementation. It should achieve MC/DC (**C3**) and run on the implementation without errors (**C4**). Two further deficiencies may occur in this phase:

**D3 Implementation Anarchy.** In violation of **C3**, the implementation contains behavior not covered by the test cases and therefore not contained in the model. Reasons for this situation include programming errors, reuse of code, use of COTS components, or undocumented functionalities.

**D4 Implementation Error.** In violation of **C4**, either code assertions are violated or some program output is inconsistent with the requirements.

If neither **D3** nor **D4** occurs, then the test suite covers the implementation and does not reveal any disagreement with its requirements.

If the evaluation step **S3** fails, we can distinguish three possible reasons: First, the requirements can be insufficient (causing **D3**) or incorrect (causing **D4**). Second, the implementation can be erroneous (causing **D4**) or contain dead code (causing **D3**). And third, the test suite generated during step **S1** can be too small to achieve coverage (causing **D3**).

The rest of this paper describes the technical realization of steps **S1-S2-S3** with our test case generation tool FShell. To integrate our approach into the processes of our industrial collaborators, we realized slope testing within the Topcased [6] framework based upon the Eclipse IDE.

- **Test Engine.** Section 2 provides a succinct summary of the FShell test case specification language FQL [4]. It focuses on *FQL filter functions*, a succinct formalism for the identification of control flow graph elements; most importantly, filter functions are used to express test goals as parts of FQL queries.

– **Tailoring Test Cases for the Detection of Slope Deficiencies.** Section 3 describes the technical realization both theoretically and with an example. Importantly, Section 3.3 introduces a formal relation between the activity diagram and the source code ("compatibility relation").
  To achieve the black-box requirement discussed above, we use filter functions to associate elements of the activity diagram with source code elements. Following the generation of a model level test suite, we use these filter functions to compose specifications for implementation level test cases from each generated model level test case. When the source code is revealed, the compatibility relation can be categorized in one of four types, i. e., a plain compatibility relation, an abstraction mapping, a decision isomorphism, or a condition isomorphism.
– **Post Mortem Assessment of Deficiencies.** In Section 3.4, we refine the discussion of deficiencies initiated above in the rigorous framework of Sections 2 and 3. Based on the distinction of the four mapping types, we systematically discuss the relationship between the deficiencies and possible reactions taken by the software developer.

The paper concludes with a case study in Section 4, and related work.

## 2 FQL & FShell

FShell's Query Language (FQL) [4, 7, 5] allows to specify a broad variety of coverage criteria, ranging from standard criteria such as statement or decision coverage [8, 9] to specifically tailored criteria used to explore isolated aspects of the program behavior. In this section we quickly survey FQL and describe an extension concerning coverage of control flow structures. FQL itself is applicable to most imperative programming languages and we provide with FShell [5] a backend for the C programming language: Given an FQL query and the corresponding source code of a program, FShell generates automatically a test suite which satisfies the coverage criterion expressed by the query on the program at hand. For example, consider the FQL query

```
> cover EDGES(@BASICBLOCKENTRY)
```

which consists of a single cover clause. This clause specifies the set of statements starting a new basic block as the set of test goals to be covered by the generated test suite.

This simple example already demonstrates the core concepts of FQL: Coverage specifications in FQL are based upon the *control flow automaton (CFA)* of the program under scrutiny. The CFA representation is similar to the control flow graph but places program locations on its nodes and statements on its edges. During query processing, the program's CFA is first filtered to obtain a *target graph:* In our case, the filter function is `@BASICBLOCKENTRY` which discards all edges which do not start a basic block. After filtering, the obtained target graph is used in a second step to generate a set of test goals. In the query above,

we take all edges of the target graph as test goals, i.e., a covering test suite must contain for each edge in the target graph a path which moves along this edge.

In variations of the above query, we can use alternative filter functions (instead of `@BASICBLOCKENTRY`) to obtain different target graphs and alternative generators (instead of `EDGES`) to build different test goal sets, as discussed in Sections 2.1 and 2.2, respectively.

FQL also allows to restrict the test cases with *path monitors* in a passing clause (discussed in Section 2.3). A path monitor is specified as a regular expression over program paths and precludes all unmatched paths as test cases. Aside from standard set theoretic recombinations of test goal sets, FQL also allows to chain test goal sets into *coverage sequences,* which require the coverage of the Cartesian product of all involved individual test goal sets, discussed in Section 2.4.

### 2.1 Target Graphs and Filter Functions

FQL refers to programs in terms of a CFA where nodes represent program locations and edges are labeled with statements. In order to select relevant parts of the CFA as target graphs FQL uses a number of filter functions.

Similar to the already discussed `@BASICBLOCKENTRY` filter function, we use the following filters to specify decision, condition, and modified condition coverage as required by DO-178B [1]: `@DECISIONEDGE` selects all edges marking the outcome of the decision of a conditional statement, `@CONDITIONEDGE` selects all edges marking the outcome of some atomic condition arising in a Boolean expression, and `@CONDITIONGRAPH` selects all parts of the CFA which are part of the evaluation of a decision of a conditional statement.

In addition to filter functions which refer to coverage relevant elements of the CFA, FQL also provides filter functions to restrict the target graph to certain areas within the source code, e. g., `@FILE("file.c")` and `@FUNC(foo)` select the CFAs corresponding to the file `"file.c"` and function `foo`, respectively. If we need to select the entire CFA, we use the identity filter `ID`.

For example, to require that each edge marking the outcome of a decision statement is covered, i.e., to require decision coverage, we use

```
> cover EDGES(@DECISIONEDGE)
```

If we would like to restrict our test suite on the source code appearing in `"file.c"`, then we would use

```
> cover EDGES(INTERSECT(@DECISIONEDGE,@FILE("file.c")))
```

where we intersect two filter functions to obtain a new one. FQL also provides set union, difference, and complementation to combine filter functions for more specific needs.

### 2.2 Test Goal Generators

Each cover clause is made up from one or more primitive test goal sets. A primitive test goal set is constructed from either states (using the keyword `STATES`),

edges (`EDGES`), paths (`PATHS`), or dependencies (`DEPS`). In case of the first three generators, each constructed test goal refers to a fragment of the CFA which must be covered by a *single* test case. For example, in `PATHS(ID,2)` the test goal set contains all possible paths which visit no state more than twice.

For the modified condition (MC) part in MC/DC, we use the fourth generator `DEPS`: First, consider the condition c in the decision **if**((a || b) && c). Modified condition coverage produces for the condition c a test goal which requires two test cases. These two test cases must evaluate c to respectively true and false – without altering the evaluation of the remaining conditions a and b while resulting in different evaluations of the overall decision of the **if** statement. Thus, (a=1, b=0, c=0) and (a=1, b=0, c=1) would cover the test goal for c, since the evaluation of c is the only difference between the two cases, while the first one evaluates the overall decision to true and the second one to false. In general, modified condition coverage requires for each atomic condition in each conditional statement two test cases which demonstrate that the decision of the statement does indeed depend non-vacuously on the condition. Thus, the two test cases must (i) lead to different outcomes of the decision, (ii) evaluate the condition in concern to different truth values, and (iii) evaluate all other conditions to the same truth value.

We omit a detailed discussion of `DEPS` and the formulation of MC/DC for space reasons. For details see [7].

### 2.3 Path Monitors

While the cover clause of an FQL query states the test goals to be covered by the requested test suite, the *passing clause* of an FQL query *restricts* the paths through the program which are eligible as test cases. Cover and passing clause are independent and can be combined freely. A passing clause specifies a *path monitor* as a regular expression on the edges of the CFA. The edges of a CFA constitute a rather large alphabet and therefore we usually identify relevant edge subsets with filters. For example, `@CALL(sort)` describes all edges in the CFA which are labeled with a call to sort. To require a test suite achieving decision coverage with test cases that invoke sort at least once, we use

```
> cover EDGES(@DECISIONEDGE) passing ID*.@CALL(sort).ID*
```

where `ID*` describes an arbitrary path through the program, and a dot denotes concatenation. Thus we ask for test cases which start with an arbitrary path through the program (`ID*`), call sort (`@CALL(sort)`), and end again with an arbitrary path through the program. As another example, consider the passing clause

```
passing (ID*.@CALL(insert).ID*)>=10.@CALL(sort).ID*
```

which requires to invoke `sort` at least once after calling `insert` at least ten times. Alternatively, with `passing COMPLEMENT(@CALL(sort))*`, one only allows test cases which never invoke `sort`.

### 2.4 Coverage Sequences

For the analysis of the interaction of different program parts, we introduced *coverage sequences* to FQL. A coverage sequence combines the test goals of state-, edge-, and path-coverage specifications into a new set of test goals based on their Cartesian product. Consider the example

```
> cover EDGES(@BASICBLOCKENTRY)->EDGES(@BASICBLOCKENTRY)
```

which requests for each *pair of basic blocks* a test case that covers both basic blocks. As another example,

```
> cover EDGES(INTERSECT(@DECISIONEDGE,@FUNC(sort)))->
  EDGES(INTERSECT(@DECISIONEDGE,@FUNC(insert)))
```

requires a test suite that features a test case for every pair of decisions occurring in `sort` and `insert`, respectively. FQL allows to construct coverage sequences of arbitrary length and to restrict the paths between the concatenated test goal sets with path monitors.

The presented material on FQL is sufficient to discuss slope testing in detail. For a precise description of FQL, please cf. [4, 7].

## 3 Slope Testing

With FQL and FShell, we can specify and solve complex coverage criteria on C programs efficiently. We use this capability to apply slope testing in an industrial context where ANSI-C is used as implementation language. The low level requirements expressed in each activity diagram are implemented by a corresponding C function. As we model function calls by corresponding links (i. e., by call behavior actions) between the individual activity diagrams, we obtain a network of interconnected activity diagrams. To relate the individual code fragments of the implementation to nodes in the activity diagram, we use annotations in the source code as described in Section 3.3. Before we discuss such technical aspects, we start in Section 3.1 with a detailed description of the slope testing process, and exemplify it in Section 3.2. Finally, in Section 3.4, we analyze the occurrences of deficiencies in the context of different mappings from activity diagrams to source code.

### 3.1 Process Description

In our tool chain, we take an activity diagram $\mathsf{Diagram}$ as model and an ANSI-C source code $\mathsf{Source}$ as implementation. Furthermore, slope testing is based upon two queries, namely $\mathsf{Query}_M$ and $\mathsf{Query}_I$, as explained below.

The model level query $\mathsf{Query}_M$ is devised by the test engineer to generate the model level test suite $\mathsf{Suite}_M$. The implementation level query $\mathsf{Query}_I$ is also formulated by the test engineer to check that the concretized test suite $\mathsf{Suite}_I$ does indeed satisfy the necessary coverage criterion, e. g., in case of software at DAL A, $\mathsf{Query}_I$ requires MC/DC. In most cases, one will start with identical

queries $\mathsf{Query}_M = \mathsf{Query}_I$. If this approach leads to deficiencies and reveals a steep slope between Diagram and Source, one can either refine Diagram to flatten the slope, refine $\mathsf{Query}_M$ to avoid some of the deficiencies, or, in case of missing traceability information or an implementation error, fix the implementation.

As already introduced in Section 1, slope testing works in three steps: In step **S1**, the test suite is *generated* from $\mathsf{Query}_M$ and Diagram. Afterwards, the test suite is *concretized* in step **S2** and *evaluated* against $\mathsf{Query}_I$ in step **S3**. If test cases cannot be concretized uniquely, then implementation poverty (**D1**) and/or implementation liberty (**D2**) occur in **S2**. If the check against $\mathsf{Query}_I$ in step (**S3**) fails, we have found an implementation anarchy (**D3**).

**S1 Test Suite Generation.** First, the test engineer must devise a model level test suite $\mathsf{Suite}_M$ which relies exclusively on the activity diagram Diagram and achieves requirements coverage. The meaning of requirements coverage is not formally defined and leaves the choice of a proper methodology to construct $\mathsf{Suite}_M$ to the test engineer. Nevertheless, $\mathsf{Suite}_M$ must achieve MC/DC upon concretization on Source at DAL A (or condition coverage at DAL B).

To assist the test engineer in the analysis of Diagram, our Topcased plug-in translates Diagram into a skeleton function in C which has a control flow identical to the structure of Diagram. This step is necessary to process Diagram with FSHELL. Then the test engineer specifies the necessary test cases with an FQL query $\mathsf{Query}_M$ to generate with FSHELL a test suite over the skeletal C code. The resulting test cases are automatically translated back into test cases for Diagram to obtain $\mathsf{Suite}_M$. A test case $\mathsf{Case}_M^i$ in $\mathsf{Suite}_M$ determines a single path through the activity nodes in Diagram.

After inspecting $\mathsf{Suite}_M$, the test engineer can either release the suite or update $\mathsf{Query}_M$ to enhance the suite until it is considered to achieve requirements coverage.

**S2 Test Suite Concretization.** During this step, each model level test case $\mathsf{Case}_M^i \in \mathsf{Suite}_M$ is concretized fully automatically to obtain an implementation level test suite $\mathsf{Suite}_I^i$ which covers all behavior described by $\mathsf{Case}_M^i$: Using FSHELL, we generate $\mathsf{Suite}_I^i$ as a test suite which covers the FQL query $\mathsf{Query}_I^i$. This query expands $\mathsf{Query}_I$ with a passing clause that requires each test case to follow the abstract path described by $\mathsf{Case}_M^i$. Hence, $\mathsf{Suite}_I^i$ covers as many test goals of $\mathsf{Query}_I$ as possible while following the abstract execution path prescribed by $\mathsf{Case}_M^i$. Note that this approach does not produce any test cases which do not cover further test goals within the behavior described by $\mathsf{Case}_M^i$. For example, it precludes for structural coverage criteria, such as MC/DC, test cases following the same execution path with other input values.

Upon concretization, we check for implementation poverty (**D1**) and implementation liberty (**D2**): If $\mathsf{Suite}_I^i = \emptyset$ holds for some $\mathsf{Case}_M^i$, then **D1** occurred, and if $|\mathsf{Suite}_I^i| > 1$ occurs, we identified an instance of **D2**.

The final concretized test suite $\mathsf{Suite}_I$ is obtained as the union $\bigcup_i \mathsf{Suite}_I^i$ of all individual test suites $\mathsf{Suite}_I^i$.

**S3 Test Suite Evaluation.** Since we obtained $\mathsf{Suite}_I$ as concretization of $\mathsf{Suite}_M$, we do not know a priori whether $\mathsf{Suite}_I$ satisfies $\mathsf{Query}_I$. Therefore, we check in this step whether $\mathsf{Suite}_I$ achieves coverage on the implementation according to $\mathsf{Query}_I$. If some test goals of $\mathsf{Query}_I$ remain uncovered, then the implementation exhibits anarchy (**D3**). Finally, the test cases in $\mathsf{Suite}_I$ are executed and unexpected program output or assertion violations result in an implementation error (**D4**).

### 3.2 Example

In this section, we exemplify slope testing throughout the steps **S1** to **S3** and demonstrate the deficiencies *implementation poverty* (**D1**), *liberty* (**D2**), and *anarchy* (**D3**), skipping only implementation errors (**D4**).

**Initial Model & Source.** We use the activity diagram Diagram depicted in the screen shot shown in Figure 1. It models a printing functionality for a linked list. The corresponding handwritten C code Source (cf. Listing 1) should realize this feature. The first action node in the model is a call behavior action that

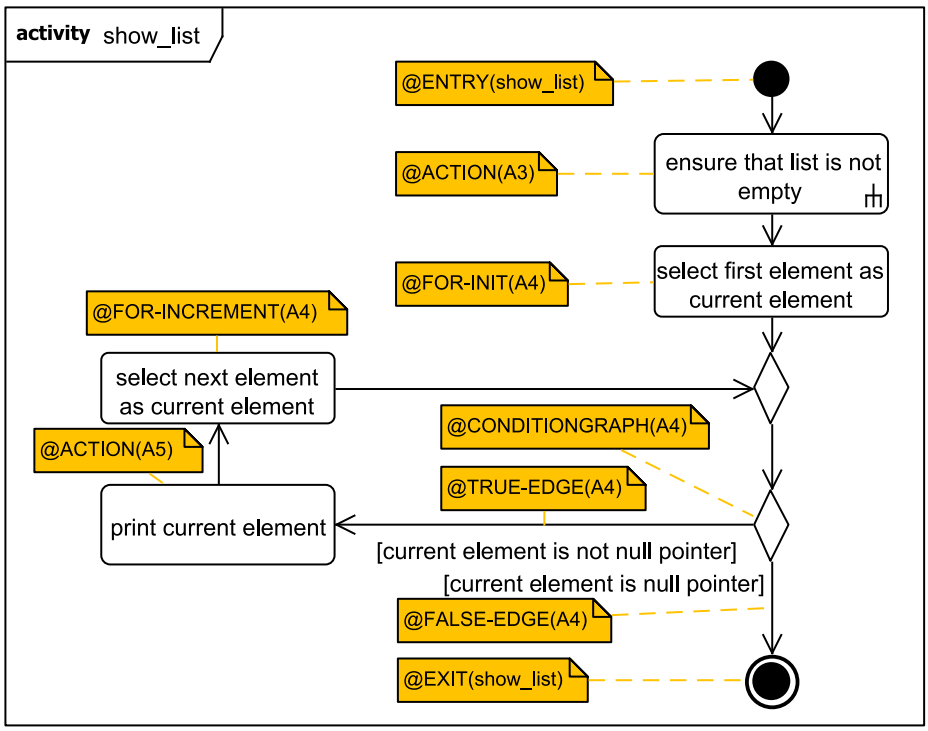


**Fig. 1.** Printing elements of a linked list

ensures that a given list is not empty. The activity diagram in Figure 2 is linked

to the call behavior action and models an assertion used for this check. There, a violated assertion is logged and a special *EXIT* node is entered that terminates program execution. The function assert in Listing 1 realizes this functionality. Model and source alike select in the next step the first element of the list as current element, print it, and set its successor as new current element. This processing is repeated until the end of the list is reached. The printing of the list is modeled with three activities that are not associated with further subdiagrams: *select head element as current element*, *print current element*, and *select next element as current element*.

Relevant elements of the activity diagrams are annotated with FQL filter expressions to identify in a given C source code the corresponding code fragments, e. g., `@ENTRY(show_list)` selects the entry of the function show_list. In our example, annotations refer to labels A0 to A5 which are contained in Listing 1. For example, the action *print current element* annotated with `@ACTION(A5)` is implemented in Listing 1 using an **if−then−else** construct which is *not* reflected in the model. Slope testing will reveal this situation.

**S1 Test Suite Generation.** In our example, we aim at achieving path coverage on the model show_list with a loop bound of 2, i. e., no node shall be visited more than twice by the same test case. Considering the activity diagram this bound is sufficient to generate test cases that do not enter the loop at all as well as test cases that do enter the loop once. Table 1 shows $\mathsf{Suite}_M$ obtained by $\mathsf{Query}_M$ = `cover PATHS(ID, 2)`.

**S2 Test Suite Concretization.** Besides $\mathsf{Suite}_M$, Table 1 gives the FQL queries $\mathsf{Query}_I^1$ and $\mathsf{Query}_I^2$ derived from $\mathsf{Case}_M^1$ and $\mathsf{Case}_M^2$. We solve these queries on Source to obtain $\mathsf{Suite}_I^1$ and $\mathsf{Suite}_I^2$ as their respective solution. Observe that the annotations in the diagrams are FQL filter function expressions used in $\mathsf{Query}_I^1$ and $\mathsf{Query}_I^2$.

The first query $\mathsf{Query}_I^1$ yields an empty test suite $\mathsf{Suite}_I^1$, i. e., we observe *implementation poverty* (**D1**). Our model does not formally model control conditions and thus the model level test case generation step **S1** did not consider

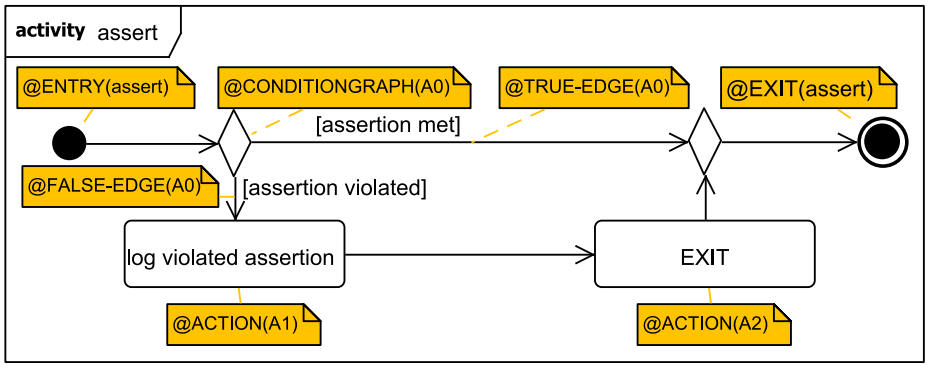


**Fig. 2.** Modeling of an assertion

```
1 void assert(int assertion, char* text) {
2 A0: if (!assertion) {
3 A1:   { log("assertion violated: %s!\n", text); }
4 A2:   { exit(1); }
5     }
6 }

8 void show_list(struct list* p_list) {
9     struct list_element* e;
10 A3: { assert(p_list->head != NULL, "head is null"); }
11 A4: for (e = p_list->head; e; e = e->next)
12 A5: { if (e->allocated) print("ALLOCATED");
13       else print("FREE");
14     }
15 }
```

**Listing 1.** Sample program

$\mathsf{Case}^1_M$: `[@ENTRY(show_list),@ACTION(A3),@ENTRY(assert),@CONDITIONGRAPH(A0),`
`@TRUE-EDGE(A0),@EXIT(assert),@FOR-INIT(A4),@CONDITIONGRAPH(A4),`
`@FALSE-EDGE(A4),@EXIT(show_list)]`

$\mathsf{Query}^1_I$:
```
cover PATHS(ID, 2) PASSING @ENTRY(show_list)>=1.@SKIP*.@ACTION(A3)>=1.
@SKIP*.@ENTRY(assert)>=1.@SKIP*.@CONDITIONGRAPH(A0)>=1.@SKIP*.
@TRUE-EDGE(A0)>=1.@SKIP*.@EXIT(assert)>=1.@SKIP*.@FOR-INIT(A4)>=1.
@SKIP*.@CONDITIONGRAPH(A4)>=1.@SKIP*.@FALSE-EDGE(A4)>=1.@SKIP*.
@EXIT(show_list)>=1
```

$\mathsf{Case}^2_M$: `[@ENTRY(show_list),@ACTION(A3),@ENTRY(assert),@CONDITIONGRAPH(A0),`
`@TRUE-EDGE(A0),@EXIT(assert),@FOR-INIT(A4),@CONDITIONGRAPH(A4),`
`@TRUE-EDGE(A4),@ACTION(A5),@FOR-INCREMENT(A4),@CONDITIONGRAPH(A4),`
`@FALSE-EDGE(A4),@EXIT(show_list)]`

$\mathsf{Query}^2_I$:
```
cover PATHS(ID, 2) PASSING @ENTRY(show_list)>=1.@SKIP*.@ACTION(A3)>=1.
@SKIP*.@ENTRY(assert)>=1.@SKIP*.@CONDITIONGRAPH(A0)>=1.@SKIP*.
@TRUE-EDGE(A0)>=1.@SKIP*.@EXIT(assert)>=1.@SKIP*.@FOR-INIT(A4)>=1.
@SKIP*.@CONDITIONGRAPH(A4)>=1.@SKIP*.@TRUE-EDGE(A4)>=1.@SKIP*.
@ACTION(A5)>=1.@SKIP*.@FOR-INCREMENT(A4)>=1.@SKIP*.
@CONDITIONGRAPH(A4)>=1.@SKIP*.@FALSE-EDGE(A4)>=1.@SKIP*.@EXIT(show_list)>=1
```

**Table 1.** Model level test suite $\mathsf{Suite}_M = \{\mathsf{Case}^1_M, \mathsf{Case}^2_M\}$ and derived FQL queries $\mathsf{Query}^1_I$ and $\mathsf{Query}^2_I$

the fact that only nonempty lists are processed by show_list. This precondition is enforced at label A3, where we assert p_list ->head != NULL, such that e is initialized at label A4 with a non-null value. Therefore, the condition of the **for** loop cannot evaluate to false and the loop body is entered at least once. As we generate C code for the activity diagram, in principle, our approach would allow to model such dependencies.

$\mathsf{Suite}_I^2$ contains two concrete test cases revealing *implementation liberty* (**D2**) in the modeling of the action *print current element*. As noted above, the **if**−**then**−**else** construct in the **for** loop of Listing 1 is not modeled. We choose to correct this situation by replacing the action *print current element* with a call behavior action that is associated with the diagram shown in Figure 3. In the source code, we add corresponding labels to associate the activity nodes with their concrete implementation (not shown in the listing).

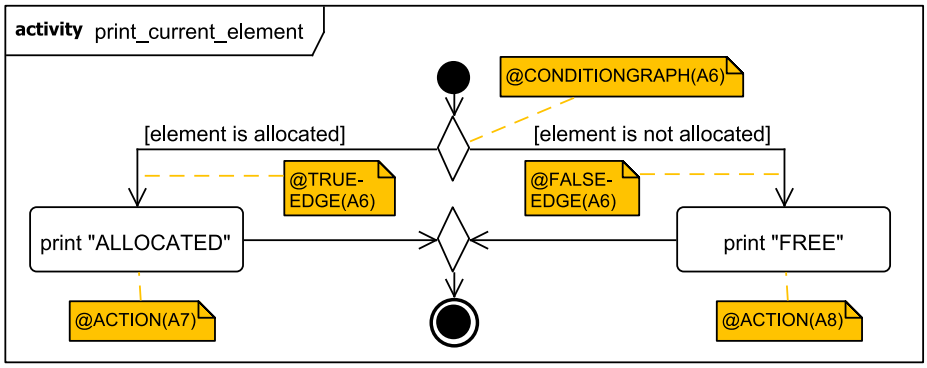


**Fig. 3.** Activity diagram of if structure

**S3 Test Suite Evaluation.** At last, during structural coverage analysis of $\mathsf{Suite}_I = \mathsf{Suite}_I^1 \cup \mathsf{Suite}_I^2$, for $\mathsf{Query}_I$ = `cover PATHS(ID, 2)`, we observe that some code in function assert is not executed. More specifically, we detected *implementation anarchy* (**D3**) in the code corresponding to an assertion failure. In this case, one would not adapt the model or the implementation but argue during integration that this assertion is never violated.

Test cases for $\mathsf{Suite}_I^2$:

```
1  struct list  l_list ;

3  struct list_element  l_element;
4  l_element.next = NULL;
5  l_element.allocated  = 1;

7   l_list .head = &l_element;
```

| Activity Node | Usage Rules |
|---|---|
| Action node | unique predecessor, unique successor |
| Decision node | unique predecessor |
| Merge node | unique successor |
| Initial node | unique initial node, no predecessor, unique successor |
| Activity final node | unique activity final node, unique predecessor, no successor |

**Table 2.** Supported Activity Nodes

```
9  show_list(& l_list );
```

**Listing 2.** Test case 1 in $\mathsf{Suite}_I^2$

```
1  struct list  l_list ;

3  struct list_element  l_element;
4  l_element.next = NULL;
5  l_element.allocated  = 0;

7   l_list .head = &l_element;

9  show_list(& l_list );
```

**Listing 3.** Test case 2 in $\mathsf{Suite}_I^2$

### 3.3 Formal FQL Query Derivation

In UML, activity diagrams are introduced to "specify the dynamic, behavioral constructs used in various behavioral diagrams" and "emphasize the sequence and conditions for coordinating lower-level behaviors" [2]. Following the modeling approach practiced by our industrial collaborators, we use restricted activity diagrams to represent the low level requirements in a manner suitable to derive a test suite and its corresponding implementation. As we rule out concurrency in our activity diagrams, our diagrams essentially describe finite state machines.

**Definition 1 (Activity Diagram).** *An* activity diagram *is a tuple* $\langle A, T, \theta \rangle$*, where* $A$ *is the set of* activity nodes*,* $T \subseteq A \times A$ *is the set of* activity edges*, and* $\theta$ *maps every activity edge starting at a decision node to a constraint.*

In Table 2, we list the supported types of activity nodes, i. e., we do not support object nodes, fork nodes, join nodes, and flow finals. We also require a number of usage rules for each supported node type, also shown in Table 2, and construct our diagrams exclusively from the control structures shown in Figure 4.

The test suite concretization **S2** makes it necessary to relate individual nodes and edges of Diagram to the corresponding parts of Source's CFA. We establish

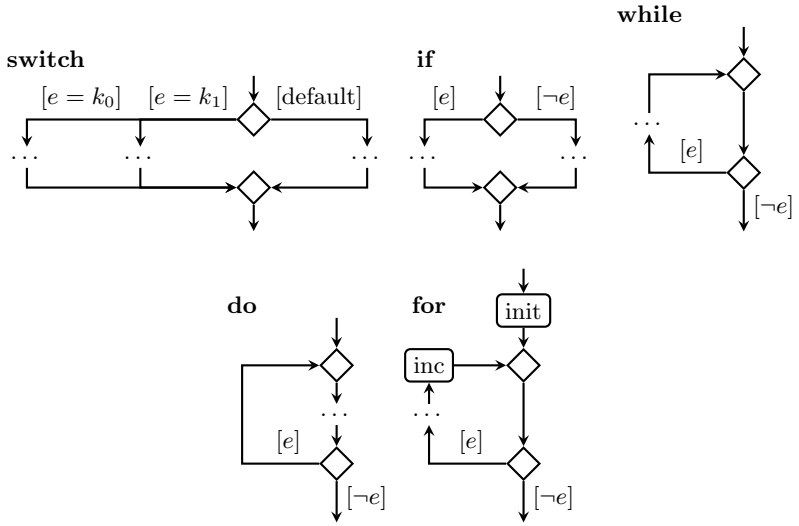


**Fig. 4.** Modeling of control structures

this relationship in two steps: First, we define for each activity node $n$ (and each relevant edge $e$) an FQL filter expression $\eta(n)$ ($\eta(e)$). This filter expression *only* depends on Diagram and is used to select the fragment of Source which is expected to implement $\eta(n)$ (or $\eta(e)$). Some trivial edges, such as edges within a basic block, remain without a filter expression and are not explicitly related to Source. Albeit defined independently from Source, the filter expressions defined by $\eta$ assume that Source is annotated with labels to mark the fragments corresponding to activity nodes and edges. Second, given Source, we categorize the resulting mapping between Diagram and Source according to their degree of accuracy, i. e., Diagram can be compatible, more abstract, or isomorphic to Source.

**Filters for Activity Nodes and Edges** For every action node $n$, we set $\eta(n) =$ `@ACTION(`$n$`)`, where `@ACTION(`$n$`)` is a filter which selects the next compound statement following label $n$. For example, for an action node `INC` described with *"print i and increment it by one"*, we use $\eta($`INC`$) =$ `@ACTION(INC)`. A typical Source matching the filter would be:

```
INC: { printf("%d\n",i); ++i; }
```

If a Diagram models a C function f, then the initial node $n_i$ in Diagram is associated with the entry of f, and its activity final node $n_f$ is associated with the exit of f, i. e., we use $\eta(n_i) =$ `@ENTRY(f)` and $\eta(n_f) =$ `@EXIT(f)`.

**Required** FQL **Filters.** Before we discuss the filter expressions associated with control structures, we introduce the necessary primitive filters: We use the filter expression `@CONDITIONGRAPH(`$n$`)` to match in Source's CFA all computations

of the conditional statement which follows immediately after the label $n$ (if no such statement exists, the filter yields the empty CFA). Likewise, `@TRUE-EDGE(`$n$`)` and `@FALSE-EDGE(`$n$`)` match in the CFA the edge corresponding to the true and false outcome of the conditional statement following label $n$.

`@FOR-INIT(`$n$`)` yields the subgraph of the CFA that corresponds to the initialization part of the **for** statement following the label $n$. Analogously, `@FOR-INCREMENT(`$n$`)` identifies the subgraph corresponding to the incrementation part.

`@CASE(`$n$`, `$d$`)` identifies the edge that is taken in case the next **switch** expression following the label $n$ evaluates to $d$. Likewise, `@DEFAULT(`$n$`)` identifies the edge that corresponds to the default case.

**Filter Expressions for Control Structures.** In case of a control structure $n$ with two outcomes (**if**, **while**, **do**, **for**, as shown in Figure 4), we define (i) a filter for the computation necessary to decide which branch to take, and (ii) a filter for those edges which correspond to one possible outcome of the decision:

$\eta(n)$ = `@CONDITIONGRAPH(`$n$`)`
$\eta(t_{[e]})$ = `@TRUE-EDGE(`$n$`)`
$\eta(t_{[\neg e]})$ = `@FALSE-EDGE(`$n$`)`

Here, $t_{[e]}$ and $t_{[\neg e]}$ are the edges starting in node $n$ which are labeled by $\theta$ with $e$ and $\neg e$, respectively.

For example, given an **if** structure named `PRECOND` with $e$ = (a || b) we use $\eta$(`PRECOND`) = `@CONDITIONGRAPH(PRECOND)` to denote the computation necessary to determine the outcome of the decision. The two outcomes of `PRECOND` are filtered for with $\eta(t_{[e]})$ = `@TRUE-EDGE(PRECOND)` and $\eta(t_{[\neg e]})$ = `@FALSE-EDGE(PRECOND)`, respectively.

Finally, in case of a **switch** control structure $n$ with outgoing edges $t_{[e=k_0]}$, $t_{[e=k_1]}$, ..., and $t_{[\text{default}]}$, the mapping is as follows:

$\eta(n)$ = `@CONDITIONGRAPH(`$n$`)`
$\eta(t_{[e=k_0]})$ = `@CASE(`$n$`, `$k_0$`)`
$\eta(t_{[e=k_1]})$ = `@CASE(`$n$`, `$k_1$`)`
...
$\eta(t_{[\text{default}]})$ = `@DEFAULT(`$n$`)`

**Relating Activity Diagrams and Source Code** Given Diagram, we determine the filter expressions $\eta(\cdot)$. These filter expressions relate an—in principle—arbitrary Source with Diagram. However, to match the filter expressions properly, Source must be annotated to link activity nodes and edges with Source's CFA.

Before checking the relation between Diagram and Source, we inline all call behavior actions which model only a part of a C function. Hence, every call behavior action refers to an activity diagram which models a complete C function.

We say that Diagram and Source are compatible, if the filter expressions described by $\eta(\cdot)$ match non-overlapping fragments in the CFA of Source. Otherwise, multiple action nodes or edges of Diagram would refer to the same code fragment.

**Definition 2 (Compatibility).** *If* Diagram $= \langle A, T, \theta \rangle$ *is an activity diagram and* Source *is some C source code, then* Diagram *and* Source *are* compatible,

*iff for all* $r \neq t \in A \cup T$ *with* $\eta(r) \neq \emptyset$ *and* $\eta(t) \neq \emptyset$, $\eta(r)$ *and* $\eta(t)$ *yield non-overlapping subgraphs of* Source*'s CFA.*

Compatibility alone assures only that we are able to process Diagram and Source with slope testing in a meaningful way. But in most cases—especially in case of certification relevant software, the relationship between Diagram and Source will be much tighter.

From a methodological point of view, we can investigate the compatibility of an activity diagram Diagram with an arbitrary source code Source, but, in practice we investigate this relationship between a model and its concrete implementation. Here, using FQL filter function expressions and source code annotations, a programmer is able to trace back source code to the model level. Such a traceability is required by DO-178B and can be monitored automatically by comparing the target graphs of the annotations before and after a modification in the source code. Therefore, we are able to handle a broad field by starting from comparing a model to some source code up to full traceability from source code to its corresponding low level requirement.

**Definition 3 (Compatible Relationships).** *Let* Diagram $= \langle A, T, \theta \rangle$ *be an activity diagram and let* Source *be some compatible C source code. Then we define the following refined relationships between* Diagram *and* Source*:*

- Abstraction Mapping: *In an abstraction mapping, every edge in the CFA which is not labeled with a* `skip` *operation is matched by some filter expressions* $\eta(r)$ *for* $r \in A \cup T$*.*
- Decision Isomorphism: *A decision isomorphism restricts an abstraction mapping further and requires additionally that the subgraph of* Source*'s CFA matched by* $\eta(r)$ *contains no branches, for all* $r \in (A \setminus A_d) \cup T$ *where* $A_d$ *is the set of decision nodes in* Diagram*.*
- Condition Isomorphism: *A condition isomorphism is defined like a decision isomorphism, but requires that* $\eta(r)$ *contains no branches for all* $r \in A \cup T$*.*

Thus, the four considered relationships between Diagram and Source form a hierarchy, with plain compatibility as weakest relationship, which is refined at increasing levels by an abstraction mapping, a decision isomorphism, or a condition isomorphism.

### 3.4 Post Mortem Deficiency Assessment

In this section we prove that, depending on the type of relationship between Diagram and Source, the deficiencies **D1** to **D3** are correlated and imply each other's occurrence. To do so, we need some further notation, introduced next: We denote with testgoals(Query, Program) the set of test goals required by the coverage criterion Query on the program representation Program, which may either be activity diagrams or some source code. The test goals covered by a test case Case are denoted covered(Case, Program).

**Definition 4 (Subsumed Query).** *Let* $\mathsf{Query}_1$ *and* $\mathsf{Query}_2$ *be two* FQL *queries. Then we say that* $\mathsf{Query}_2$ subsumes $\mathsf{Query}_1$ *and write* $\mathsf{Query}_1 \subseteq \mathsf{Query}_2$*, if*

$$\mathsf{covered}(\mathsf{Query}_1, \mathsf{Program}) \subseteq \mathsf{covered}(\mathsf{Query}_2, \mathsf{Program})$$

*holds for every program representation* Program*. If the subset relation is a strict one, we say that* $\mathsf{Query}_2$ strictly subsumes $\mathsf{Query}_1$ *and write* $\mathsf{Query}_1 \subset \mathsf{Query}_2$*.*

If $\mathsf{Query}_1 \subseteq \mathsf{Query}_2$ holds, then every test suite satisfying $\mathsf{Query}_2$ also satisfies $\mathsf{Query}_1$. Thus, this definition requires that both $\mathsf{covered}(\mathsf{Query}_1, \mathsf{Diagram}) \subseteq \mathsf{covered}(\mathsf{Query}_2, \mathsf{Diagram})$ *and* $\mathsf{covered}(\mathsf{Query}_1, \mathsf{Source}) \subseteq \mathsf{covered}(\mathsf{Query}_2, \mathsf{Source})$ hold. Below, we are mostly interested in the setting when $\mathsf{Query}_I \subseteq \mathsf{Query}_M$ holds: If the relation between Diagram and Source is a true abstraction (i. e., does not describe a condition isomorphism), then the test engineer will have to compensate the lack of modeling precision with a more rigid model level coverage criterion, i. e., $\mathsf{Query}_I \subset \mathsf{Query}_M$ will be necessary.

To analyze the reason for occurring deficiencies, we need to assure that the test goals derived from Diagram and Source are comparable. Therefore, we introduce the query Structural which subsumes all queries which impose only test goals referring to the control structures present in both, Diagram and Source: $\mathsf{Structural} = \mathsf{BB} \cup \mathsf{DC} \cup \mathsf{MCC} \cup \mathsf{PC}$. Therein, BB, DC, MCC, and PC denote basic block, decision, multiple condition, and path coverage (cf. [10, 11]).

Furthermore, we assume that Diagram is at least an abstraction of Source, since otherwise, some parts of Source would not even be modeled in abstract terms and remain completely unrelated with Diagram.

**CASE I: General Abstraction.** If Diagram is an abstraction of Source, then implementation liberty (**D2**) can occur, since every model level action potentially refers to arbitrary constructs at the implementation level. Thus, it is possible that more than one implementation level test case is required to cover the abstracted control flow. For this reason, implementation liberty becomes an approximate measure for the degree of abstraction between Diagram and Source: The more abstract Diagram is, the more implementation level test cases in $\mathsf{Suite}_I^i$ are on average necessary to cover all behavior described by test case $\mathsf{Case}_M^i$ at model level.

Intuitively, if implementation poverty (**D1**) occurs and Diagram abstracts Source, then some part of the implementation must remain uncovered, i. e., implementation anarchy (**D1**) occurs. In the next lemma, we make this intuition precise. We say that a model or implementation level test suite Suite, which satisfies some coverage criterion Query, contains a redundant test case $\mathsf{Case} \in \mathsf{Suite}$ if $\mathsf{Suite} \setminus \{\mathsf{Case}\}$ is still satisfying Query.

**Lemma 1.** *If* Diagram *is an abstraction of* Source *and* $\mathsf{Suite}_M$ *contains no redundant test case, then implementation poverty (**D1**) implies implementation anarchy (**D3**) for all queries* $\mathsf{Query}_I = \mathsf{Query}_M \subseteq \mathsf{Structural}$*.*

The converse of Lemma 1 does not hold for general abstraction mappings, i. e., an instance of implementation anarchy (**D3**) does not imply implementation

poverty (**D1**). This is true, since anarchy can also occur in lieu of liberty for two reasons: (i) A model level test case $\mathsf{Case}^i_M$ with a non-empty concretization $\mathsf{Suite}^i_I$ covers some further action which refers to some unreachable code in the implementation. (ii) A fraction of the code referred by some covered action is only reachable, if this action is approached in some specific manner. In the latter case, the abstraction appears to be an unsuitable choice, since the code referred by the action is not independent from the remaining code.

*Remark 1.* If implementation anarchy (**D3**) occurs without poverty (**D1**), then some action in Diagram refers to dead code in Source, or the reachability of the referred code crucially depends on the path reaching the action – indicating an unsuitable abstraction from Diagram to Source.

Therefore, if implementation anarchy (**D3**) occurs without poverty (**D1**), then the reason should be tracked down and either eliminated or thoroughly justified with a suitable rationale (following Remark 1). Otherwise, if implementation anarchy (**D3**) and poverty (**D1**) occur together, we start eliminating implementation poverty (**D1**) for the following reason: Since FSHELL never produces test suites with redundant test cases, Lemma 1 is applicable and hence we know that each instance of poverty (**D1**) entails an occurrence of anarchy (**D3**). To eliminate poverty (**D1**), either Diagram or $\mathsf{Query}_M$ should be refined as to ensure that all test cases in $\mathsf{Suite}_M$ can be concretized.

At the end of this process, implementation liberty (**D2**) without poverty (**D1**) or anarchy (**D3**) occurs, and therefore the remaining slope is purely a result of the abstraction between Diagram and Source: The steeper the slope, the stronger the dependence of $\mathsf{Suite}_I$ on Source. Depending on the involved certification regulations, it might be necessary to refine Diagram to lower the slope.

**CASE II: Condition Isomorphism.** In case of condition isomorphism, implementation liberty (**D2**) cannot occur because of the same control flow structure in Diagram and Source (shown below in Lemma 2). Making practically unrestraining assumptions on the involved queries, we show that implementation poverty (**D1**) occurs iff anarchy (**D3**) occurs (Lemmata 1 and 3). In consequence, the absence of implementation poverty (**D1**) in the context of a condition isomorphic Diagram and Source is sufficient to show that none of the deficiencies (**D1**) to (**D3**) occurred (Theorem 1).

Therefore, in case of condition isomorphism, one should refine Diagram or $\mathsf{Query}_M$ until no implementation poverty (**D1**) occurs. Note that no coverage check against implementation anarchy (**D3**) is necessary.

**Lemma 2.** *If* Diagram *is condition isomorphic to* Source*, then implementation liberty (**D2**) cannot occur during slope testing for any pair of* FQL *queries* $\mathsf{Query}_M$ *and* $\mathsf{Query}_I$ *with* $\mathsf{Query}_I \subseteq \mathsf{Structural}$.

**Lemma 3.** *If* Diagram *is condition isomorphic to* Source*, then implementation anarchy (**D3**) implies implementation poverty (**D1**) for every pair of queries* $\mathsf{Query}_I \subseteq \mathsf{Query}_M \subseteq \mathsf{Structural}$.

**Corollary 1.** *If* Diagram *is condition isomorphic to* Source, $\mathsf{Suite}_M$ *contains no redundant test case and* $\mathsf{Query}_I = \mathsf{Query}_M \subseteq$ Structural *holds, then implementation poverty (**D1**) occurs iff implementation anarchy (**D3**) occurs.*

**Theorem 1.** *If* Diagram *is condition isomorphic to* Source*, then the absence of implementation poverty (**D1**) shows that none of the deficiencies (**D1**) to (**D3**) occurred, for all queries* $\mathsf{Query}_I \subseteq \mathsf{Query}_M \subseteq$ Structural*.*

**Case III: Decision Isomorphism.** In case of decision isomorphism, we cannot rule out liberty since a model level test case passing a decision **if** (a || b) through its true-edge has two choices in the unfolded CFA: Either evaluating a to true and leaving through the true-edge, or evaluating a to false and continuing with the evaluation of b resulting in a true outcome as well.

However, if we rule out all test goals referring to individual conditions and assume decision isomorphism, then no test goal at the implementation level can possibly refer to a CFA structure not present at the model level. Therefore, we can restate the results and proofs of the preceding section (Lemmata 2 and 3, their Corollary 1 and Theorem 1) for decision isomorphism by replacing Structural with $\mathsf{BB} \cup \mathsf{DC}$ and condition by decision isomorphism.

Therefore, all coverage according to every criterion subsumed by $\mathsf{BB} \cup \mathsf{DC}$ can be enforced with an Diagram and a query $\mathsf{Query}_M$ that produces no implementation poverty (**D1**). If condition-based coverage criteria are used, one must revert to the procedure for general abstraction mappings, i. e., one must check for implementation anarchy (**D3**) as well and eliminate both, implementation poverty (**D1**) an anarchy (**D3**) until liberty (**D2**) alone remains.

## 4 Case Study

We present two case studies—a memory manager and an engine controller—that underline the practical applicability in an industrial context. In both case studies we apply slope testing using our Eclipse plug-in as described in Section 3. We performed our experiments on a 2.33 GHz AMD64 system with 16 GB RAM. Characteristics of our benchmarks and a summary of our results are shown in Table 3.

As a case study from the avionics domain we consider a memory manager for helicopter software. The memory manager is implemented in 526 lines of ANSI-C code[5] and will be a core component of several other applications to be deployed as part of a pilot assistance and mission planning system. The purpose of the memory manager is to avoid memory fragmentation caused by dynamic memory allocation. It provides an API to prepare a memory manager object and to provide individual chunks in the previously acquired memory area to application components.

[5] Source lines of code (SLOC) are measured using David A. Wheeler's SLOCCount tool.

Taken from the automotive domain, we studied the software for an engine controller. The implementation of 3449 lines of ANSI-C code was built from a MATLAB/Simulink model using automated code generation. We manually inserted labels into the source code to establish traceability.

Comparing the execution times of stages **S1**–**S3** for the memory manager and the engine controller, as shown in Table 3, we observe that our approach scales well with the implementation size. The relatively long execution times of stage **S3** are primarily due to the compiler used to build executables with execution trace logging and analysis. We expect to replace this part by alternative means in future releases of our framework.

| | Memory M. | Engine C. |
|---|---|---|
| SLOC | 526 | 3449 |
| Activity Diagrams | 21 | 2 |
| Action Nodes per AD | $\leq 13$ | 9 and 17 |
| Decision Nodes per AD | $\leq 2$ | 4 and 7 |
| Loops | $\leq 1$ | 0 |
| Model-Level TC | 27 | 171 |
| Time **S1** [s] | 19 | 7 |
| Time **S2** [s] | 60 | 1880 |
| Time **S3** [s] | 1050 | 8893 |
| Poverty | 1 | 169 |
| Liberty | 10 | 1 |

**Table 3.** Summary of experimental results

**Memory Manager.** The requirements for the memory manager yielded 21 activity diagrams, 17 of which are implemented as C functions, two are part of these functions, and the remaining two are macros. Both, activity diagrams and source code were annotated as described in Section 3.3. The level of detail of the activity diagrams suggested that a condition isomorphism should be achieved. As the experiments revealed, however, this was not always the case (see below). Because of their simple structure, we applied path coverage at model level. Several models, however, contain loops. We therefore bound the number of permitted repetitions by two, i. e., we use $\mathsf{Query}_I = \mathsf{Query}_M =$ `cover PATHS(ID, 2)`. In total, we obtained a model level test suite of 27 test cases. The initial concretization revealed several errors in design and coding: In one diagram, the order of activities was not correctly reflected in the implementation. This implementation anarchy (**D3**) resulted in a set of activities not being covered by the test suite and one model level test case with no matching concretization, causing implementation poverty (**D1**). Some action nodes were implemented using loops, which were consequently not properly covered by the test suite. This implementation liberty (**D2**) required changes in the activity diagrams to gain a 1-1 correspondence.

**Engine Controller.** Compared to the memory manager, the implementation of the engine controller is about seven times as large. Due to the nature of the domain, however, only abstract activity diagrams were available. The two diagrams contain four and seven decisions, respectively. At implementation level, however, 252 decisions were found. We added annotations to establish an abstraction mapping and estimate the slope. Here, we also used $\mathsf{Query}_I = \mathsf{Query}_M =$ `cover PATHS(ID, 2)` and obtained 161 model level test cases, whereas their concretization only produced two implementation level test cases. The reason for this implementation poverty (**D1**) was found through code inspection: Parts of the generated engine controller code proved to be unreachable. As the activity diagrams were abstractions that did not reflect all decisions, implementation liberty (**D3**) was expected. We obtained nine different concrete test cases for one model level test case, witnessing the steep slope between the abstract model and its implementation.

## 5 Related Work

There are several approaches to generate test cases for UML models: In the tradition of automata theoretic methods, the most common [12] approaches employ UML statecharts [13, 14] and interaction diagrams [15], respectively. To answer a strong industrial demand, we generate test cases based on activity diagrams.

Test case generation approaches based on activity diagrams are introduced in [16–18]. Chen et al. [17] propose a method to randomly generate test cases for Java programs. Since their technique is based on randomness, a good coverage cannot be guaranteed. In [16] a coverage directed approach for test case generation is proposed that uses the model checker NuSMV. As the authors themselves mention, the proposed method is vulnerable to the state space explosion problem. Therefore we assume this approach to be inadequate for practical use in an industrial context.

Kundu and Samanta [19] present an extension of [16, 18]. They overcome restrictions with regard to loops and concurrent system behavior. We do not deal with concurrency – however, in our application domain of safety-critical embedded systems, most individual software components are still designed and implemented in a non-concurrent fashion. Kundu and Samanta use activity diagrams to model use cases and are therefore much more abstract. The generated test cases are apparently not executable without additional processing.

Considering the level of sophistication, *Automated Gray-Box Testing* [20] is much closer to our work than the ones mentioned above. This approach combines black-box testing on model level with white-box parametrized unit testing. They generate parametrized unit tests for an extended version of activity diagrams and instantiate these tests via white-box test case generation. For their case study they report high implementation coverage, but, do not state their coverage criterion.

## References


1. RTCA DO-178B: Software considerations in airborne systems and equipment certification (1992)
2. OMG: UML 2.0 Superstructure Specification. Technical Report ptc/04-10-02, Object Management Group (2004)
3. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: systematic test case generation for dynamic analysis and measurement. In: CAV. (2008) 209–213
4. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: A Precise Specification Framework for White Box Program Testing. Technical Report TUD-CS-2009-0148, TU Darmstadt (2009)
5. Holzer, A., Tautschnig, M., Schallhart, C., Veith, H.: Query-driven program testing. In: VMCAI. (2009) 151–166
6. Pontisso, N., Chemouil, D.: Topcased combining formal methods with model-driven engineering. In: ASE. (2006) 359–360
7. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Dependency Coverage Criteria with FQL. Technical Report TUD-CS-2009-0149, TU Darmstadt (2009)
8. Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to softwaretesting. Software Engineering Journal **9**(5) (1994) 193–200
9. Myers, G.J.: The Art of Software Testing. John Wiley and Sons (1979)
10. Ball, T.: A theory of predicate-complete test coverage and generation. In: FMCO. (2004) 1–22
11. Ntafos, S.C.: A comparison of some structural testing strategies. IEEE Trans. Software Eng. **14**(6) (1988) 868–874
12. Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: a systematic review. In: WEASELTech. (2007) 31–36
13. Weißleder, S., Schlingloff, B.H.: Deriving input partitions from UML models for automatic test generation. In: MoDELS Workshops. (2007) 151–163
14. Chevalley, P., Thévenod-Fosse, P.: Automated generation of statistical test cases from UML state diagrams. In: COMPSAC. (2001) 205–214
15. Nayak, A., Samanta, D.: Model-based test cases synthesis using UML interaction diagrams. SIGSOFT Softw. Eng. Notes **34**(2) (2009) 1–10
16. Chen, M., Mishra, P., Kalita, D.: Coverage-driven automatic test generation for UML activity diagrams. In: GLSVLSI. (2008) 139–142
17. Chen, M., Qiu, X., Li, X.: Automatic test case generation for UML activity diagrams. In: AST. (2006) 2–8
18. Chen, M., Qiu, X., Xu, W., Wang, L., Zhao, J., Li, X.: UML activity diagram-based automatic test case generation for java programs. The Computer Journal **52**(5) (2009) 545–556
19. Kundu, D., Samanta, D.: A novel approach to generate test cases from UML activity diagrams. Journal of Object Technology **8**(3) (2009) 65–83
20. Kicillof, N., Grieskamp, W., Tillmann, N., Braberman, V.A.: Achieving both model and code coverage with automated gray-box testing. In: A-MOST. (2007) 1–11

## A Proofs

*Proof (of Lemma 1).* If we face implementation poverty (**D1**), then there exists a model level test case $\mathsf{Case}_M^i \in \mathsf{Suite}_M$ with $\mathsf{Suite}_I^i = \emptyset$. Since $\mathsf{Suite}_M$ contains no redundant test case, $\mathsf{Case}_M^i$ covers some test goal which is not covered by any other test case in $\mathsf{Suite}_M$. Because of $\mathsf{Query}_M \subseteq \mathsf{Structural}$, this test goal refers to a state, edge or path in the CFA of Diagram.

This test goal refers to a potentially larger subgraph in the CFA of the implementation: Since we have $\mathsf{Query}_I = \mathsf{Query}_M$, this subgraph in Source's CFA gives rise to at least one implementation level test goal (in case of a condition isomorphism between Diagram and Source, it is a single test goal, in case of a true abstraction, there will be more test goals). We fix one such test goal.

Hence $\mathsf{Suite}_I$ must cover this test goal to satisfy $\mathsf{Query}_I$. But since all test cases in $\mathsf{Case}_I \in \mathsf{Suite}_I$ must follow some test case $\mathsf{Case}_M \in \mathsf{Suite}_M \setminus \{\mathsf{Case}_M^i\}$—and none of them covers the test goal in question—this test goal must remain uncovered – causing implementation anarchy (**D3**).

*Proof (of Lemma 2).* For the sake of contradiction, assume that an implementation liberty occurs, i. e., there exists a $\mathsf{Case}_M^i \in \mathsf{Suite}_M$ which is concretized to $\mathsf{Suite}_I^i$ with $|\mathsf{Suite}_I^i| > 1$.

Then there must exist two distinct test cases $\mathsf{Case}_I \neq \mathsf{Case}_I'$ in $\mathsf{Suite}_I^i$ which both implement $\mathsf{Case}_M^i$ but hit different test goals, i. e., $\mathsf{Case}_I$ and $\mathsf{Case}_I'$ both concretize $\mathsf{Case}_M^i$ but $\mathsf{covered}(\mathsf{Case}_I, \mathsf{Source}) \neq \mathsf{covered}(\mathsf{Case}_I', \mathsf{Source})$ holds.

But since $\mathsf{Query}_I \subseteq \mathsf{Structural}$ produces only states, edges, and paths from the CFA as test goals, the inequality in the covered test goals implies that the execution paths induced by $\mathsf{Case}_I$ and $\mathsf{Case}_I'$ deviate at some point—although being abstracted to $\mathsf{Case}_M^i$. Thus, there must exist a branch which is mapped into a single action—but this is a contradiction to the assumption that Diagram is condition isomorphic to Source.

*Proof (of Lemma 3).* If implementation anarchy (**D3**) occurs, then we have found some test goal which is not covered by $\mathsf{Suite}_I$. Since $\mathsf{Query}_I \subseteq \mathsf{Structural}$ holds, this test goal must refer to a state, edge, or path in the CFA of Source. Because of the condition isomorphism, this CFA structure must exist in Diagram as well, and since $\mathsf{Query}_I \subseteq \mathsf{Query}_M$ holds, it must be covered at model level as well. Since $\mathsf{Suite}_M$ satisfies $\mathsf{Query}_M$ (since it is generated in step (**S2**) to achieve coverage), there exists a covering test case $\mathsf{Case}_M^i$ at the model level. But as the concretization is missing, it must have been lost through concretization, i. e., $\mathsf{Suite}_M^i = \emptyset$ holds and implementation poverty (**D1**) occurred.

*Proof (of Corollary 1).* The Corollary combines Lemmata 1 and 3.

*Proof (of Theorem 1).* Follows from Lemmata 2 and 3: Lemma 2 rules out implementation liberty (**D2**). Lemma 3 states that implementation anarchy (**D3**) implies implementation poverty (**D1**) and therefore, the absence of poverty (**D1**) implies the absence of anarchy (**D3**).