



# From diagnosis to diagnosability: axiomatization, measurement and application <sup>☆</sup>

Yves Le Traon <sup>a,\*</sup>, Farid Ouabdesselam <sup>b</sup>, Chantal Robach <sup>c</sup>, Benoit Baudry <sup>a</sup>

<sup>a</sup> IRISA, Campus de Beaulieu, Université de Rennes I, 35042 Rennes Cedex, France

<sup>b</sup> LSR-IMAG, BP 72, 38402 St. Martin d'Hères Cedex, France

<sup>c</sup> LCIS-ESISAR, 50 Av. Barthelemy de Laffemas, BP 54, 26902 Valence Cedex 9, France

Received 12 February 2001; received in revised form 10 April 2001; accepted 5 October 2001

## Abstract

Classical views on testing and their associated testing models are not dealing with the question of fault repairing but only focus on fault detection. Diagnosis consists of determining the nature of a detected fault, of locating it and hopefully repairing it. Correlatively, the only standardized quality factors implied in the detection/repair aspects of software engineering are testability and maintainability: those quality factors are misleading since they do not pinpoint this question of the location/repairing effort, that can be identified under the concept of diagnosability. This paper is thus concerned with diagnosability, its definition and the axiomatization of its expected behavior. The paper aims at:

- introducing and analysing diagnosability as a significant and complementary dimension of software testability,
- producing a high-level definition and axiomatization of a diagnosability measurement generic enough to be adapted to various software paradigms: this property-based approach serves as a measurement “specification”, independent on the application context and thus reusable,
- detailing a diagnosability measure dedicated to data-flow software and especially test strategies impact on diagnosis and testing effort (from measure implementation to case study),
- illustrating the reuse of the high-level axiomatization to the specific question of measuring the impact of assertions (or contracts for a designed by contract OO system) on diagnosis effort and preciseness.

Throughout the paper, the concepts are illustrated on a case study provided by an industrial partner. At last, the reusability of the axiomatization is illustrated by proposing a measure of the impact of assertions (or contracts in a design by contract approach) on global software diagnosability. Main lessons concern both the diagnosability significance as a quality factor and the interest of an axiomatization-based methodology for building trustable software measurement.

© 2002 Elsevier Science Inc. All rights reserved.

## 1. Introduction

A failure may be observed during the software development as well as the maintenance stage. Given an

occurrence of a failure, diagnosis requires a set of additional symptoms in order to determine the faulty part of the system which causes the detected failure. *Diagnosis* is thus defined as the task which consists in locating faulty components of a system when a failure is detected.

Test and diagnosis are complementary activities: while the test objective is the fault detection, diagnosis aims at pointing out fault localizations for correcting them. Diagnosis is thus an inescapable task which may be extremely difficult depending on the nature of faults and software complexity. From theoretical and

<sup>☆</sup> This work was supported by two grants from AÉROSPATIALE: no. 504 22455 (1993) and RAF-000488 (1997).

\* Corresponding author. Tel.: +33-2-9984-2568; fax: +33-2-9984-7171.

E-mail addresses: [yves.le\\_traon@irisa.fr](mailto:yves.le_traon@irisa.fr), [yletraon@irisa.fr](mailto:yletraon@irisa.fr) (Y. Le Traon), [farid.ouabdesselam@imag.fr](mailto:farid.ouabdesselam@imag.fr) (F. Ouabdesselam), [chantal.robach@esisar.inpg.fr](mailto:chantal.robach@esisar.inpg.fr) (C. Robach), [benoit.baudry@irisa.fr](mailto:benoit.baudry@irisa.fr) (B. Baudry).

pragmatic point of views, test and diagnosis are generally considered separately. Indeed, there is neither logical nor experimental correlation between testing and diagnosis efforts. Even if a software has a trend to reveal its faults during testing (Voas, 1995), there is no reason that these detected faults will be easy to locate and to correct. The problem is that an important part of the validation effort is spent in locating and correcting faults, and not only in detecting them. This confusion remains for the specification-design phase since testability is the predictive quality factor which is traditionally connected to the validation cost: *testability* is a predictive test effort and test efficiency estimate and not a diagnosis effort estimate. Testability and *diagnosability*—roughly defined here as the property of faults to be easily and precisely located—are never correlated. In this paper, we claim that both testability and diagnosability must be taken into account and separately estimated at an early design stage. The improvement of a design in terms of testability can be obtained by a degradation of diagnosability: gaining in testability should not be researched to the detriment of the diagnosis effort. Despite its real impact on the development process cost, the diagnosability attribute is not commonly defined in the literature: this software attribute is not practically distinguished from testability. To be well defined, and intuitively well understandable, the diagnosis process has to be studied first and then diagnosability defined with respect to the diagnosis process.

This paper aims at providing predictors of diagnosability applicable at an early design stage and that evolve with the design refinement steps. From a designer point of view, two progressive levels of preciseness are proposed: global analysis for detecting a potential diagnosability weakness of a design and local one for precisely locating the parts of the design that cause this detected weakness. The measurements are thus hierarchically decomposed in global and local measurements. Global measurement detects design weaknesses by defining an expected diagnosis effort that is global to the design. When a weakness is detected, local measurements related to each of the design components are provided for determining the parts of the system which cause the diagnosability weakness.

To be consistent with software metrication theory (Shepperd and Ince, 1993; Fenton, 1991; Briand et al., 1996), we decompose each step for elaborating the proposed metrics: from the intuitive meaning of the diagnosability attribute to the axiomatization of the corresponding predictors. To define a rigorous metric, the expected behavior of the diagnosability attribute is expressed in this paper in terms of several operations on designs. This axiomatization serves as a basis for checking that measurements are consistent with expressed intuitive properties. It also serves as a high-level set of behavioral requirements for the measured quality

factor, that can be reused/adapted to other problem domains as it will be illustrated.

The article opens with a short discussion on the diagnosis process and the software attributes which have an impact on diagnosability. Section 3 is devoted to data flow design characteristics and testing, since our approach to diagnosability is first dedicated to such designs. Section 4 concentrates on the definition of diagnosability and its axiomatization. Section 5 defines diagnosability measurements. The sixth section shows how diagnosability analysis is carried out on a selected piece of avionics software. Last section illustrates the reuse/adaptation of the presented diagnosability axiomatization to a software paradigm other than data-flow designs: designed by contract OO systems (and assertions based procedural programming). The impact of contracts on system diagnosability can thus be estimated and thus the interest of designing by contract.

## 2. Diagnosis practices in the software domain

To analyze the diagnosability attribute, one needs to understand the methods used for locating faults in the software when detected.

A first category of diagnosis techniques is based on program slicing techniques. These techniques focus on the software code at the unit and integration levels. Various slicing methods exist (Weiser, 1982, 1984; Kamkar, 1995; Korel, 1997) which basically consist in extracting from the program a set of statements which can be executed independently (this corresponds to a *slice* of the program). A direct fault localization process focuses on a faulty value of a variable  $x$  for a statement  $s$ . Slicing is used for debugging as follows: all statements that might affect the value of  $s$  are computed according to Weiser's algorithm. Two sets are also attached to each statement for helping fault localization: a set of variables that contains relevant variables whose values can affect the variables of each statement and a set of predicates that may indirectly affect this statement. An oracle has also to be defined at each step to know whether or not the fault is due to the set of statements included in the slice under consideration. If the slice is a single statement, then the fault is located. If the slice corresponds to a procedure, the same approach may be applied recursively to locate more precisely the fault into the body of the procedure. The benefits of these approaches may be summarized as follows: slices are automatically calculated, each slice is a part of a program which can be independently executed and the diagnosis is precise enough for locating a fault in a statement. The main limitation of this technique is its cost in terms of human effort. Indeed, each slice implies the determination of an oracle and, because the slices have no simple functional meaning, it often needs human intervention.

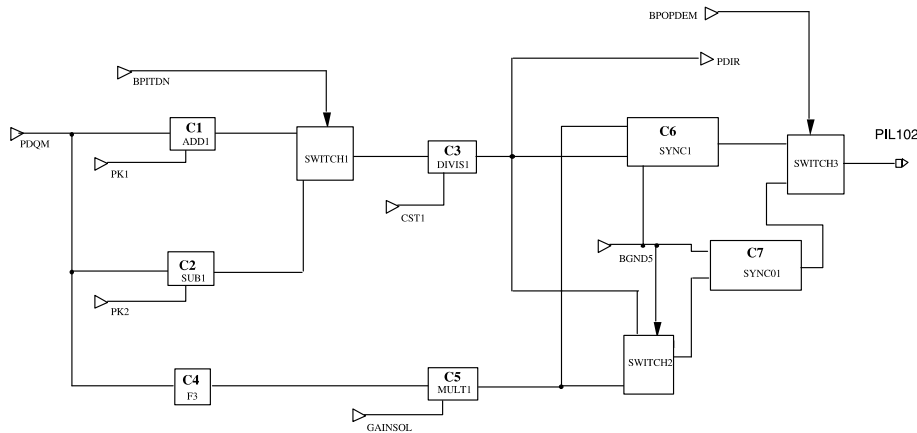


Fig. 1. SAO diagram.

This direct localization process does not allow to reduce the scope of the possible suspected statements in the slice.

As an indirect way of locating faults, (Agrawal et al., 1995) details a pragmatic cross-checking strategy based on the comparison between two dynamic slices. The difference between two slices is called a “dice”. To locate a fault we consider that we have two slices (A and B): if one of them (B) gives a correct result, and the other (A) gives a faulty one, the fault area will suspect the statements of the dice of (A minus B). Hence the effort of searching in the whole slice A is saved and the time of diagnosis reduced. The generalization of this idea to “*n*-slices” consists of performing some cross-checking between information resulting from several test executions. Such systematic cross-checking of test results and executed slices leads to semi-automated diagnosis strategies (Khalil et al., 1998).

Besides, another classical way for locating faulty statements consists in inserting assertions in the program for detecting some internal faulty state during execution. The systematic use of assertions before and after procedure calls may be very efficient for detecting and locating faults. However, the effort for defining and inserting assertions may be important because it implies a good understanding of the internal meaning of the procedure and expected values of the data. Some works have focused on the way of inserting assertions when needed in the program, with testability criteria (Voas, 1995). The issue of diagnosability in an assertion-based design approach such as the “design by contract” methodology (Meyer, 1992; Baudry, 2000) is discussed in Section 7.

Confronted with the problem of diagnosis, which remains a non-automated task, it is useful to appraise the probable difficulty of locating faults in the software beforehand. Such predictor estimate is called diagnosability, and provides a way of improving the design quality.

### 3. Design context and a formal model

The objective of a diagnosability analysis is to estimate the expected effort and preciseness of the localization process. The associated measurement cannot be defined *in abstracto*, since faults are always revealed in a given testing context. As we measure diagnosability as a predictive estimator, and since the measurement depends on the testing context, we need to qualify the testing context at the design level. The abstraction of the testing context is called a *test strategy*. In this section, we briefly present the design and testing contexts.

#### 3.1. Design context

We are concerned with data flow specification languages such as Lustre (Caspi et al., 1987; Ouabdesselam and Parissis, 1996) and SADT (Ross and Schoman, 1977). Their application results in designs in terms of interconnected boxes or operator nets. As an example, Fig. 1 presents a piece of software using a method called SAO, which stands for (in English) “computer-aided system specification”. SAO is a home method of AÉROSPATIALE’s.<sup>1</sup> Each diagram represents a system functionality and corresponds to a network of interconnected boxes which can be decomposed into lower-level diagrams. The result is a sequence of hierarchically organized diagrams: the highest level diagram represents the whole system, whereas each lower-level diagram shows a limited amount of details. The graphics-based language which describes these diagrams consists in two simple primitive constructs: boxes and arrows. Each box is labeled. A box corresponds to a transform which is operated on the data carried by the inward *flows*, and whose results are the data carried by

<sup>1</sup> AÉROSPATIALE is the French partner of the AIRBUS consortium.

the outward flows (the flows go rightward). Boxes symbolize either built-in operators (AND, OR, SWITCH, ...) or user-defined functions. A user-defined box may be further decomposed into a collection of diagrams, or it merely stands for a piece of code. For the SWITCH primitive, the transform is performed under the control of a logical condition, which is characterized by an arrow.

In the following, the underlying model is assumed to catch all the possible flows. A flow is defined in this paper as a set (possibly a sequence) of components which can be exercised together, from a set on inputs to a set of outputs. It is presented in the following section.

### 3.2. The information flow model

The diagnosability measurement is carried out on a model of the design. In this section, the complete selected model is described which serves both for diagnosability and testability analysis. So the presented model is richer than needed for diagnosability measurement: it is also defined for measuring information losses through the software. However, any model which enables the extraction of the main possible execution paths through the design would be sufficient. A testability-oriented presentation of the selected model can be found in (Le Traon et al., 2000; Le Traon and Robach, 1997).

#### 3.2.1. Formal model objectives

From the test point of view, the key task is the identification of basic functional computational units. We argue that data flow analysis on data flow designs can be used to automatically recognize functions and to ensure that they have been tested.

Thus, the model we are looking for must have three properties:

- be a good basis for structural testing with respect to control flow-based test case selection criteria and data flow-based criteria alike,
- bring basic software functions to light, out of the design specifications, so that these functions can be independently tested,
- be applicable at any level of the hierarchical design down to the code level.

Of course, functional components may be implicit or anonymous, and not necessarily encapsulated in a separate functional description, in particular not in a single layer of the hierarchical design.

The model is both suited for testability and diagnosability analysis. In this paper, we focus on diagnosability analysis and only few details will concern testability. Indeed, modeling for testability analysis of data flow software amounts to constraining this model

so as to facilitate both the identification of the main subfunctions together with their components, and the representation of information propagation through these components. Finally, as it will be discussed in the following section, diagnosability does not require any supplementary constraint on the model. It is sufficient to identify indistinguishable sets of components into a given testing context (see next section for diagnosability analysis and indistinguishability attribute).

The *information flow model* has been borrowed from (Robach and Wodey, 1989); it has been successfully applied to hardware high-level descriptions and software avionics designs. This model is defined to be the single modeling means, regardless of the different forms which the software can take from design to code.

#### 3.2.2. Information transfer graph

The principle of modeling information transfers consists in representing control and data flow aspects on the same graph. This is done by means of a bipartite directed graph, called *information transfer graph*, in which two types of nodes occur: the modules which are associated with computations, and the transfers which characterize the mode of information transmission between these modules (the interconnection of modules models the capability of transmitting information from one module to another one). For any computation node there can be only one information flow coming in or going out. So, an indegree (respectively outdegree) greater than one reflects a situation in which a selection among the entering (respectively exiting) flows has to be operated. Thus, the computation nodes play the role of or-nodes, whereas the transfer nodes are similar to and-nodes. The information carried by the edges arriving at a node is all that is necessary to activate this node. Similarly, every arc leaving a node contains all the information issued by the node.

Three basic modes of information transfer are necessary to encode SAO diagrams and Pascal programs (see Fig. 2):

- the junction mode: the data from the source modules  $S_1, \dots, S_n$  are all needed for the destination module  $D$  to be exercised;

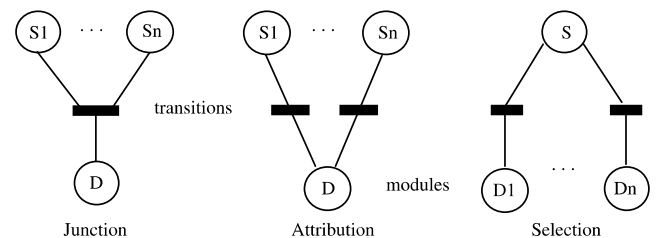


Fig. 2. Information transfer modes.

- the attribution mode: in order to be activated, the destination module  $D$  needs an information from only one of the source modules  $S_1, \dots, S_n$ ;
- the selection mode: the information issued by the source module  $S$  is sent to only one of the destination modules  $D_1, \dots, D_n$ .

The construction process of the information transfer graph associated with a SAO diagram relies on a small set of rules and schemas which are overviewed and illustrated in the following section.

*Information transfer graph of a SAO diagram. Prime graphs:* Fig. 3a presents the basic model associated with simple operators (AND, OR, ...) and with assignments. Fig. 3b depicts the information transfer graph of a SWITCH operator whose inputs are  $E_1$  and  $E_2$ , and whose control condition is  $C$ .  $E_1$ ,  $E_2$  and  $C$  are either primary inputs of the diagram or local variables which denote themselves the result of other computations. Fig. 3c corresponds to a simple delay operator, where the new value of the output depends on the previous output result. Those graphs are called prime graphs, since they cannot be decomposed trivially by the processes of sequencing and nesting. The graph in Fig. 3a represents also a user-defined binary function not yet detailed. It is an instance of the prime graph schema associated with  $n$ -ary ( $n > 0$ ) functions. Loops corresponding to iterations and repetitive treatments are not modeled in a data flow approach: such a processing is considered as a black box and modeled by the Fig. 3a prime graph. Finally, a box or a diagram with several outputs is viewed as several sub-functions operating in parallel; its modeling leads to as many distinct graphs except when the sub-functions share some sequence of statements or the same control structure. The later case is exemplified by the information transfer graph shown in Fig. 4. The models of more complex diagrams are obtained by decomposition into prime graphs.

*Concatenation (sequencing) of graphs:* The modeling of sequences of operators and function calls requires that several individual graphs be connected together. Attention must be paid to the way information is passed on between SAO boxes. Fig. 3a is representative of the model based on a junction transfer node; it expresses the

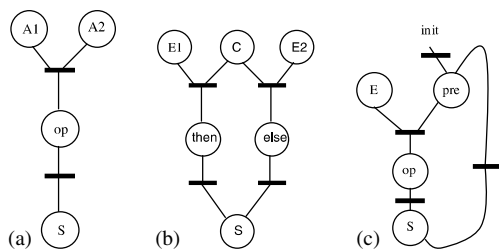


Fig. 3. Prime graphs: (a)  $S := op(A_1, A_2)$ , (b) if  $C$  then  $S := E_1$  else  $S := E_2$  and (c)  $S := op(E, init)$  then  $op(E, pre(S))$ .

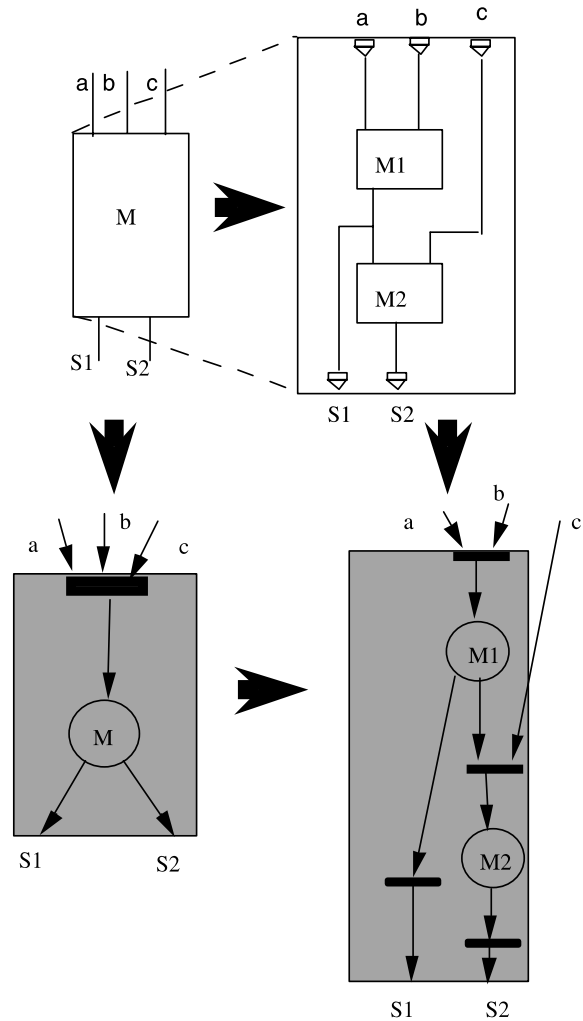


Fig. 4. Instantiation of a box and their corresponding ITGs.

sequencing of boxes in which the information needed for the execution of a box (modeled as  $op$ ) must be collected from several sources (represented by  $A_1$  and  $A_2$ ). In Fig. 3b the connection of the module  $C$  to the modules called ‘then’ and ‘else’ through a selection transfer node illustrates the case where boxes are joined in sequence with a piece of information sent to several boxes the execution of which are mutually exclusive. Indeed, this graph models a situation in which the boxes represented by  $E_1$  and  $C$  (respectively  $E_2$  and  $C$ ) act in parallel and are composed in sequence with the box modeled as ‘then’ (respectively ‘else’).

*Nesting of graphs (instantiation of nodes):* Nesting is performed either to replace the sub-graph associated with a SAO box by the graph of the corresponding detailed design diagram, or to develop the body of a loop (the expression on which the delay operator is applied). The nesting rules are not presented here since they are rather straightforward. Nesting is simply done by replacing edges by local models, and taking care of the information transfer mode.

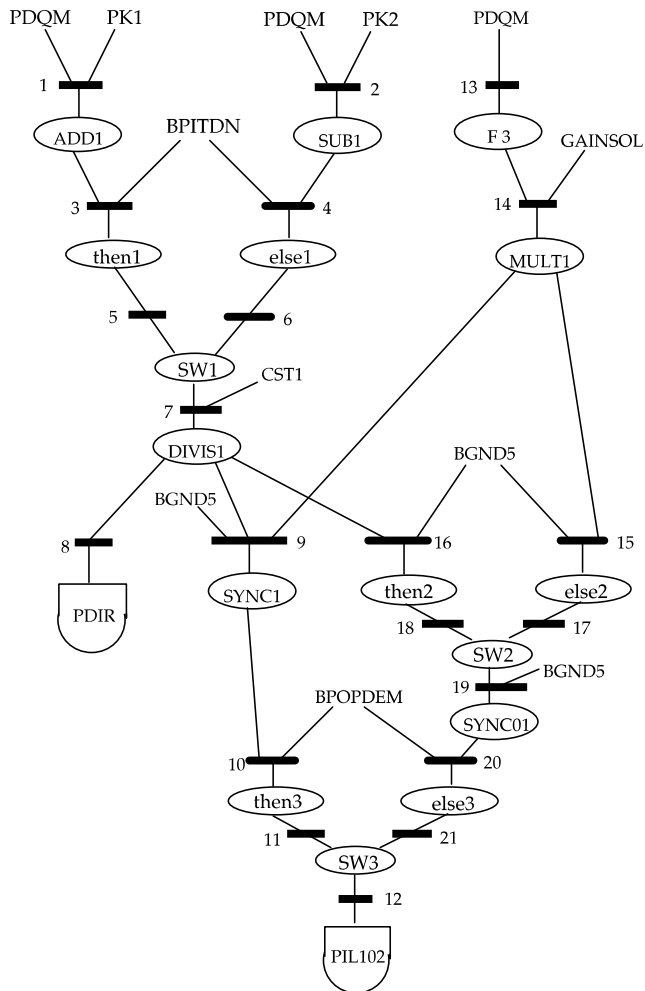


Fig. 5. Information Transfer Graph. *Note:* For sake of readability, intermediate nodes as then, else and SW(itch) have been added (in a way, they stand for dummy assignments to local implicit variables). Ovals represent modules, semi-circles represent output variables; input variables are only denoted by their names.

The process of decomposition by sequencing and nesting associates the diagram shown in Fig. 1 with the graph presented in Fig. 5. Clearly, this graph embodies all of the control information and the data dependencies.

### 3.2.3. Information flow computation

The information transfer graph groups the set of paths along which information is forwarded, from a subset of inputs towards a subset of outputs. These paths characterize the global software functions. In the context of a functional specification language, a notion of flow in the information transfer graph is proposed: a flow is an information path from inputs to one output through a set of modules. Thus, a flow characterizes a sub-function of the specification since it corresponds to one of the expressions the values of which define the output variable of concern.

A flow is computed from one output back to its connected inputs. A flow is fully defined by the set of transfer nodes passed through. Flows are such as:

- all predecessor modules and entries of a junction node in a flow also belong to this flow,
- at least one of the predecessor modules or entries of an attribution node in a flow belongs to this flow.

Each loop generates two flows: one going at least once through the loop, the other not entering the loop. The problem of infeasible flows is not tackled. In the example under consideration (see Fig. 1), seven flows are identified and listed below. They define all possible ways to compute the values of the two output variables PDIR and PIL102. Each flow  $F_i$  is noted as a set {list of all exercised transfer nodes | output variable}.

$$F_1 = \{1, 3, 5, 7, 8 | \text{PDIR}\}$$

$$F_2 = \{2, 4, 6, 7, 8 | \text{PDIR}\}$$

$$F_3 = \{1, 3, 5, 7, 9, 10, 11, 12, 13, 14 | \text{PIL102}\}$$

$$F_4 = \{2, 4, 6, 7, 9, 10, 11, 12, 13, 14 | \text{PIL102}\}$$

$$F_5 = \{12, 13, 14, 15, 17, 19, 20, 21 | \text{PIL102}\}$$

$$F_6 = \{1, 3, 5, 7, 12, 16, 18, 19, 20, 21 | \text{PIL102}\}$$

$$F_7 = \{2, 4, 6, 7, 12, 16, 18, 19, 20, 21 | \text{PIL102}\}$$

### 3.3. The testing context

Being given the design context and the associated formal modeling for testability, the way the test is planned and carried out can be defined and discussed from a diagnosis point of view.

*Test strategy:* A test strategy results in an ordered set of flows which have to be exercised on the software design. A test strategy corresponds to a test data selection criterion (i.e. the rules to select test cases) and a test data adequacy criterion (i.e. the rules used to determine whether or not testing is complete). The selection criterion is to visit (cover) every component in the model at least once by executing the selected flows in a certain order. The adequacy criterion tells us to stop testing when the specified list of flows has been exercised.

We consider two families of test strategies: incremental and cross-checking strategies.

*Incremental (bootstrapping) test strategy:* An incremental test strategy consists in progressive test and coverage of the design. It is incremental since a new flow is tested only if faults detected in a previous flow are fixed. So an incremental test strategy may be defined as an ordered list of flows to be exercised and tested.

*Cross-checking test strategy:* A cross-checking test strategy consists in exercising a set of flows and then collecting all test results before trying to locate and fix detected faults. It can be defined as a (non-ordered) set of flows to be exercised and tested.

Several strategies can be considered into these families of test strategies. In this paper, we restrict our study to four significant strategies called *All-Paths*, *Multiple-Clue*, *Start-Small* and *Start-Big*. We argue that they are representative from the two main manners of locating detected faults and that they are precise enough for studying diagnosability.

The *Start-Big* and *Start-Small* strategies are incremental methods. In *Start-Small*, flows are ordered from the smallest, covering the fewest components, to the largest. In *Start-Big*, the model is covered with the minimum number of flows. *Multiple-Clue* and *All-Paths* strategies are cross-checking strategies which differ from one another by the selection criterion. In *Multiple-Clue*, a minimum number of flows is selected in order to cover every component of the model while the *All-Paths* criterion is the coverage of all the simple paths through the model. In this paper, we only consider pure cross-checking strategies: they are only applicable for the case of a single fault, since otherwise the analysis becomes impossible. In (Khalil et al., 1998), a larger spectrum of cross-checking strategies is studied, which are based on relaxed assumptions to allow multiple-faults location. All the strategies may be used for maintenance purposes, where it can be reasonably assumed that the software (and particularly embedded software) only has a very small number of faults revealed at a time. What makes the use of cross-checking plausible during maintenance is the fact that the probability is high for a failure to be due to one main error: cross-checking procedures are effective to perform a first diagnosis to locate it. In a maintenance context, the difficulty is to be able to replay the scenario leading to the failure and thus obtaining exploitable data for performing cross-checking. Except for *All-Paths* strategy, they all combine the same test data selection criterion and the same test data adequacy criterion.

*Example.* On our main example: *Start-Small* =  $(F_1, F_2, F_6, F_5, F_3)$ ; *Start-Big* =  $(F_1, F_5, F_7)$ ; *All-Paths* =  $(F_1, \dots, F_7)$ ; *Multiple-Clue* =  $(F_1, F_3, F_5, F_7)$ .

### 3.3.1. Considerations about test strategies and diagnosability

The property of a software to be more or less diagnosable must be analyzed by taking into account the testing context, and more precisely the various possible test strategies. Bridging the gap between the design into a given testing context and a diagnosability predictor implies an abstraction and a simplification of the diagnosis process. We consider that diagnosis is performed under the following reasonable assumptions:

- test cases are produced by testers and used as a basis for fault localization,
- for each test case, the oracle verdict on whether a fault has been detected is available,
- at least one test case has detected a fault,
- components exercised by each test case are known.

A test case exercises a given flow and includes input test data and an oracle function. Based on these assumptions, we estimate the diagnosis preciseness induced by the set of test cases (hereafter called test set) as well as the effort for locating a detected fault.

For example, if the test set is reduced to one single element, the only part of the system which can be suspected is the sequence of components exercised by this test case. No other information allows the tester to reduce this group of suspected components. If we have more than one test case, then we must distinguish the case of incremental test strategies (where fault repair is performed at each step) from cross-checking ones.

### 3.3.2. The case of cross-checking test strategies

In Fig. 6a, there is no simple way of deciding that the faulty component is component 2. All components are equally suspect. The localization effort is thus related to the size of the sets of suspected components. The bigger the suspected sets are, the less precise the diagnosis is. If two test cases  $C_1$  and  $C_2$  both detect a fault, the group of components which are suspected is the intersection of the set of components exercised by  $C_1$  and the set of components exercised by  $C_2$ . In Fig. 6b, the component 8 cannot be distinguished from its bordering components 7 and 9, but this set of components may be

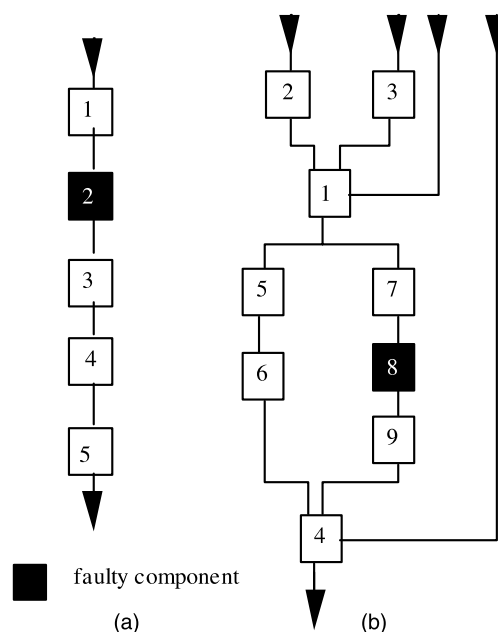


Fig. 6. Diagnosis and indistinguishable components.

isolated from the rest of the model depending on the flow selection. These intuitive considerations can be expressed by three statements:

- the diagnosability is composed of the localization effort and the diagnosis preciseness,
- the localization effort and the diagnosis preciseness are closely connected,
- the diagnosability depends on the component isolation capacity in the structure by applying a test strategy covering this structure.

Indeed, it is intuitively more difficult to distinguish among ten components the faulty one than to determine it among two components. Applying the same reasoning, the diagnosis preciseness obtained by a test strategy is higher if the fault is located among two components rather than among ten components. The underlying attribute expressing the localization of faulty components among a set of suspected components is called *indistinguishability*. Components in a set of suspected components are thus *indistinguishable* from a diagnosis point of view. The formal definition of the indistinguishable components depending on the family of test strategies is given in Appendix A. To conclude, *diagnosis effort* and *preciseness* can be both related to the number of indistinguishable components in which the faulty component has to be localized.

### 3.3.3. The case of incremental test strategies

Similarly, applying an incremental strategy generates some groups of indistinguishable components. Through the following simple example, we show that diagnosis preciseness depends on the ordering of the flows (the test strategy) and that indistinguishable components are produced at each step. Let us consider the first steps of two incremental test strategies used on the graph of Fig. 6 (in bold are represented the new tested components):

#### Strategy 1

- Step 1: flow (1, 2, 4, 5, 6)  
 Step 2: flow (1, **3**, 4, 7, **8\***, **9**)  
 Step 3: ...

#### Strategy 2

- Step 1: flow (1, 2, 4, 5, 6)  
 Step 2: flow (1, 2, 4, 7, **8\***, **9**)  
 Step 3: ...

At each step of the testing process, a new flow is tested. At a given step, if no fault has been detected in a given flow, each component in this flow is assumed to be correct. Then, in the example above, strategy 1 and strategy 2 should detect the fault at step 2. For strategy 1, the suspected set of components will be {3, 7, 8, 9} while for strategy 2 the suspected set will only be {7, 8, 9}. So, the

diagnosis will be more difficult using strategy 1. At each step of an incremental strategy, all new tested components are suspected without having any simple way of distinguishing the faulty one among them. So, the notion of indistinguishable components does exist for incremental test strategies as for cross-checking ones. Moreover, diagnosability depends on the chosen test strategy.

### 3.3.4. Remarks

In the case of dataflow designs, interoperability/interactions do not occur since there are no callbacks: failures may be the result of faulty connections between two components. The fault is not clearly located into a component but a connection between several components must be suspected. As an example, a missing component causes a failure which will be located in the connection where it is missing. For all these cases, the location process is not modified: when a set of components is suspected, the location implicitly includes the verification of connections. A failure that occurs after a particular component is executing may be caused by a fault in another (or several others) components. This vicious case is difficult to analyze only with incremental strategies: the cross-checking strategies proceed by correlating the causes while incremental ones assume that at a given step all already tested components are correct. In that case, the location process only points out the component affected by the fault: a slicing technique can be applied for determining the faulty components.

To summarize, the notion of diagnosis effort and preciseness is first related to the indistinguishability notion for a given test strategy. The same concepts (“indistinguishable components” and “indistinguishability sets”) have a meaning for both cross-checking and incremental test strategies.

## 4. Diagnosability definition and axiomatization

In this section we establish a formal basis for diagnosability measurement, by refining the intuitive diagnosability definition into a set of behavioral axioms, derived from intuitive considerations (Le Traon et al., 1998). Any diagnosability measurement suited to dataflow designs must be coherent with the expressed axioms or question the intuition. Since the intuitive aspects of diagnosability have been already discussed, diagnosability and its related attributes are defined. Its properties and the expected behavior of the associated measurements under various design operations are then detailed.

### 4.1. Informal definition

**Definition (Diagnosability).** Diagnosability expresses the localization effort as well as the preciseness allowed by a test strategy on a given design.

The effort depends on the selected test strategy. Some components, for a given test strategy, are indistinguishable from each other.

**Definition (Indistinguishable components).** Two components are indistinguishable from each other if they are always exercised together.

**Definition (Indistinguishability set (IS)).** An indistinguishability set for a test strategy  $S$  corresponds to a set of indistinguishable components for  $S$ .

**Definition (Local diagnosability ( $\delta$ )).** The local diagnosability of a component  $C$  for a strategy  $S$  is the probable effort needed for determining if  $C$  is faulty when  $S$  detects a fault.

**Definition (Global diagnosability ( $\Delta$ )).** The global diagnosability of a design for a strategy  $S$  is the probable effort needed for pointing out the faulty component, knowing that  $S$  detects a fault.

#### 4.2. Axiomatization

The axioms formalize the essential expected properties of a diagnosability measurement. They define what designs should be comparable and generic characteristics these measures must satisfy. Based on the definitions given in the previous paragraph, three sets of axioms are provided: global axioms, local axioms and axioms linking global and local measures. All the following axioms are formally defined in Appendix B. They constitute the basis of the theoretical evaluation which was carried out (Le Traon, 1997). The theoretical evaluation precedes empirical evaluation since it is less time-consuming and more appropriate to show that the model is internally consistent: it is used to detect that no pathological structures exist for which the model produces inappropriate behavior.

Measures profiles:

$$\delta : \text{Component} \times \text{Design} \times \text{Test\_Strategy} \\ \rightarrow \text{Integer over } [1 \dots + \infty],$$

$$\Delta : \text{Design} \times \text{Test\_Strategy} \rightarrow \text{Real over } [1 \dots + \infty].$$

##### 4.2.1. Local diagnosability axioms

**LDA1 (Component comparison).** All components of a design are comparable in terms of local diagnosability.

**LDA2 (Relationship between indistinguishable components).** Indistinguishable components have the same local diagnosability.

The following axioms concern the intuitive behavior of the measures under some design operations: design concatenation, component instantiation into a design (design refinement) and addition of observation points, which are presented in Fig. 7:

- refining a box or a diagram. This operation is called *Detail*.
- connecting a box or a diagram to another box or diagram. This operation is called *Concatenate*.
- substituting a detailed diagram for a box. This operation is called *Instantiate*.
- adding an exit to a box or a diagram. This operation is called *Add\_Output*.

**LDA3 (Design concatenation).** The local diagnosability of a component cannot increase when this design is concatenated to another one.

In a dataflow context, it has to be noted that concatenation does not add extra output points from the point of view of a component in a design. If a design  $D_1$  is concatenated to a design  $D_2$ , its internal components cannot be isolated using the outputs of  $D_2$  in a better way than using directly the outputs of  $D_1$ —even if  $D_2$  outputs are more numerous than  $D_1$  outputs—since:

- $D_2$  inputs depend on  $D_1$  outputs: the flows coming at  $D_2$  outputs are traversing  $D_1$  outputs first,
- $D_1$  outputs are closer from any component inside  $D_1$  than  $D_2$  outputs: the flow is easier to interpret when it is closer from the source of information.

Conversely, the same reasoning can be applied for  $D_2$  components.

**LDA4 (Component instantiation).** In a design, the diagnosability of a component cannot increase by instantiation of another component.

**LDA5 (Addition of an observation point).** The diagnosability of a component can only increase by addition of an observation point to the enclosing design (i.e. the diagnosis effort is lower).

##### 4.2.2. Global diagnosability axioms

**GDA1 (Design comparison).** Two designs are always comparable in terms of diagnosability.

**GDA2 (Design concatenation).** The diagnosability of a design obtained by concatenation of two designs  $D_1$  and  $D_2$  is lower than the diagnosability of the most diagnosable design among  $D_1$  and  $D_2$ .

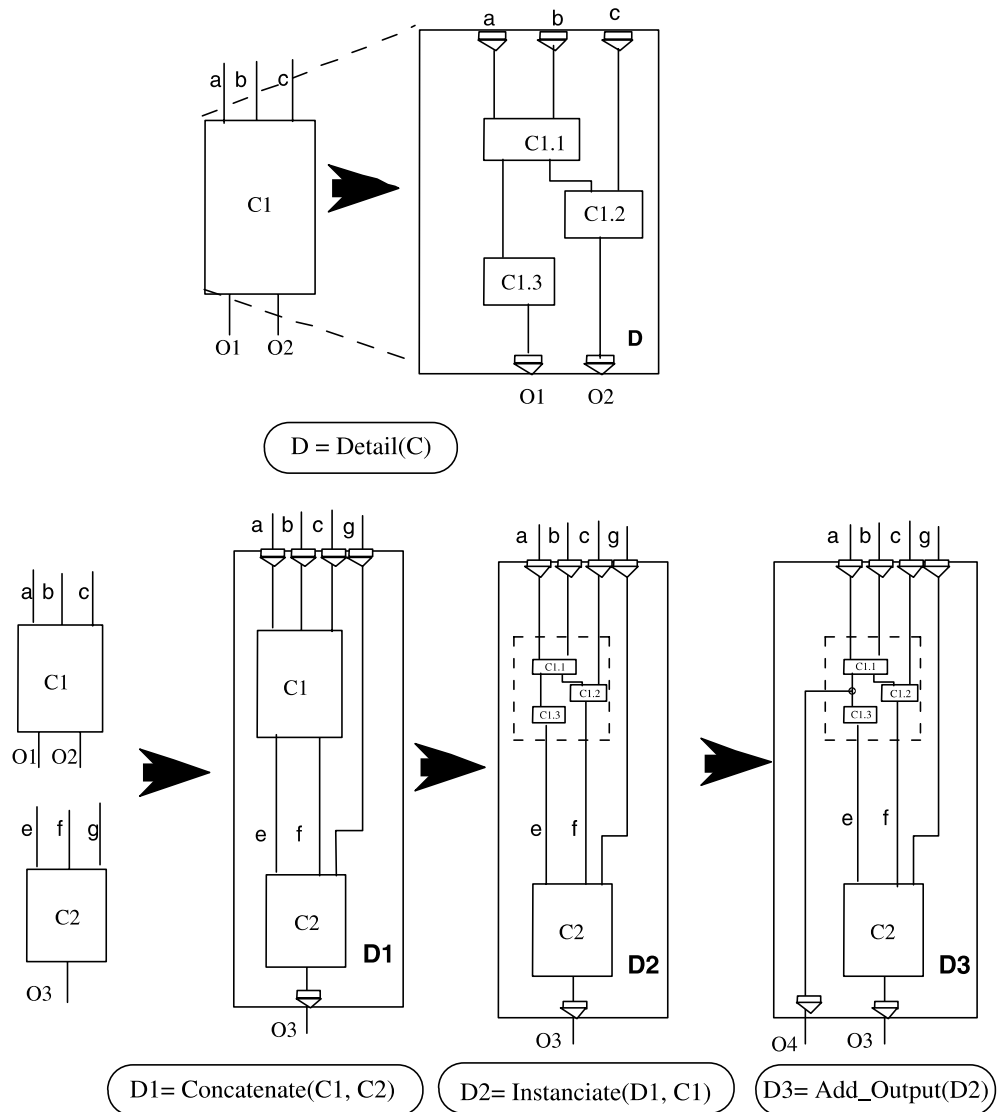


Fig. 7. Designs operations.

**GDA3 (Observation point addition).** For any design, its global diagnosability cannot decrease by addition of an observation point.

#### 4.2.3. Inter-level axioms

The knowledge of local diagnosability measures must be sufficient to deduce design global diagnosability.

**IDA1 (Inter-level relationship).** An injective function relates local diagnosability to global diagnosability.

The role of axiomatization is threefold: it provides an unifying framework for evaluating the different measures which can be proposed, thus helps one separate the search for a measurement from the statement of the intuition (the expected behavior), and brings a reusable consensual basis for applying it to other problem do-

main (such as design by contract as presented in Section 7).

## 5. Diagnosability measurements

### 5.1. Local diagnosability measurement

**Definition (Size of an indistinguishability set).** The size of an indistinguishability set is equal to the cardinality of this set. The smallest size is equal to 1; it is associated with the case when the suspected set is made up of only one component.

**Definition (Local diagnosability ( $\delta$ )).** The local diagnosability of a component is equal to the size of the indistinguishability set to which it belongs.

Let  $C_i$  be a component of a design  $D$ , where  $C_i$  is included in the indistinguishability set IS for the strategy  $S$ . We have  $\delta(C_i, D, S) = |\text{IS}|$ .

### 5.2. Global diagnosability measurement

Let  $p$  be the size of the indistinguishability set under consideration, and  $n$  be the total number of components. Let  $P_{C_i}$  be the probability that the component  $C_i$  is faulty and let IS be an indistinguishability set such as  $\text{IS} = \{C_{\text{IS}j}/j \in [1 \dots p]\}$ . The probability that the detected fault is located into IS in function of the probability  $P_j$  for each component  $C_{\text{IS}j}$  to be faulty is  $P_{\text{IS}} = \sum_{j=1}^p P_j$ .

The probable size of the indistinguishability set in which the detected fault is located is easily deduced, based on the probability to suspect each of the components. The problem is to give a probability for suspecting components. Two solutions may be envisaged:

- each component may be equally suspected. We have thus  $P_i = 1/n$  and  $P_{\text{IS}} = p/n$ ,
- a weight  $m_i$  is associated with each component corresponding to its complexity. Indeed, the more complex the component is the more probable it can produce a fault. We have:

$$P_i = \frac{m_i}{\sum_{i=1}^n m_i} \quad \text{and} \quad P_{\text{IS}} = \frac{\sum_{j=1}^p m_{\text{IS}j}}{\sum_{i=1}^n m_i} = \sum_{j=1}^p P_{\text{IS}j}.$$

The first solution enables the analysis of designs without any knowledge of the component structure: it is a particular case of the second solution since it consists in considering that all the component weights are the same. The second solution allows the distinction between complex components considered as more suspect and simple or reliable components. In the remaining of the paper we assume that all components have the same weight.

**Definition (Global diagnosability ( $\Delta$ )).** Global diagnosability is the probable effort needed to locate a faulty component with a strategy  $S$  knowing that  $S$  detects a fault. This effort is equivalent to the probable size of the indistinguishability set in which the fault must be located. It is mathematically expressed as follows:

Let  $D$  be a design,  $S$  a test strategy,  $\text{IS}_q$ ,  $q \in [1 \dots \text{nsets}]$  the indiscernability sets for  $S$ :

$$\Delta(D, S) = \sum_{q=1}^{\text{nsets}} |\text{IS}_q| P_{\text{IS}_q}.$$

### 5.3. Considerations on the empirical validation of diagnosability measures

To be the most meaningful, the diagnosability estimate should be related to the actual diagnosis effort,

thus to a measure of this effort. It appears that there is no simple and objective way for this purpose. A first technique which comes to mind is to measure the time spent from the error detection to error localization. However, this measurement is misleading since the diagnosis effort is intertwined with the tools used during the debugging process. Another way is to count the number of operations needed for making the faulty component observable. Several techniques are available: break points with a debugger, assertions, code slicing or observation points insertion. This proves that the diagnosis effort depends on the debugging context, and therefore, to be meaningful, any evaluation of this effort should represent a trend rather than an absolute value. In this case, the intuition which underlies the diagnosability as formalized in the previous sections, also applies to the diagnosis effort. To confirm this intuition, relevant experiments should be conducted on a large scale and should be statistically well-founded.

### 5.4. Global test difficulty

Testability and diagnosability are uncorrelated factors. A design may be more difficult to test than another one but may lead to a better diagnosis. For example, the introduction of observation points increases the diagnosis preciseness but implies new flows to be tested. Consequently, improving a design will often be a trade-off between testability and diagnosability improvements. Thus, we have to define testability and a way of measuring it.

**Definition (Testability).** We define the quality factor “testability” as the ease of testing a piece of software design using structural testing strategies. This easiness is both an intrinsic property of the design (thus a proper characteristic of the product) and a property correlated to the test strategy which is used (thus, a joint characteristic of the product and the process).

At the global level, testability is influenced by three parameters: the global controllability, the global observability (presented in (Le Traon and Robach, 1997) for measures and (Le Traon et al., 2000) for axiomatization), and the overall difficulty of checking the validity of the execution results after the application of a given test strategy: since we focus on this last parameter, we use a paths counting measure called global test difficulty.

**Definition (Global test difficulty (GTD)).** Let  $D$  be a design,  $S$  a test strategy,  $\{f_i, i \in [1 \dots \text{nflows}]\}$  the selected flows for  $S$ , The global test difficulty of  $D$  for  $S$  is defined as the sum of the selected flows test costs (FTC):

$$\text{GTD}(D, S) = \sum_{i=1}^{\text{nflows}} \text{FTC}(f_i).$$

The flow test cost is defined as a function of the exercised components:

$$\text{FTC}(f) = \Phi(C_i, \dots, C_n).$$

As a first estimate we consider that  $\text{FTC}(f) = |f|$ .

In SAO designs, a SWITCH component is counted as two components to take into account the test of else/then subfunctions. This counting measure has also been adapted and applied to the problem of measuring hardware/software partitioning impact on test cost with VHDL design descriptions (Al-Hayek et al., 1997).

## 6. Case study

The whole approach to diagnosability has been applied to two significant industrial systems related to either the Airbus A320 aircraft or the Ariane 5 space rocket. In this section, two diagrams are separately studied which come from a decomposition of a more complete design. This piece of software design is embedded in an on-board control unit integrated in a flight management system and is composed from two concatenated diagrams (case study 1 and case study 2). Case study 1 details the example which is being treated since Section 2. Case study 2 concerns the second diagram while case study 3 addresses the complete design. For sake of conciseness, we focus on the global diagnosability measurement without detailing the way of locating the diagnosability weaknesses into a particular design.

### 6.1. Case study 1

The application of the Start-Small strategy leads to the results presented in the Table 1. The results for all the strategies are given in Table 2 as well as the global test difficulty values. It can be noted that the All-Paths strategy does not allow a better diagnosis than the Multiple-Clue one (Multiple-Clue flow selection allows to reach an optimal preciseness and the lowest diagnosis effort in a cross-checking approach). So, the global test difficulty for Multiple-Clue is always lower than for All-Paths. In the case of cross-checking strategies, Multiple-Clue appears to be an interesting solution to reduce the test effort without reducing diagnosability. Regarding

Table 1  
Application of Start-Small to case study 1

Step	Flow	Tested components	$\delta$	$P_{Is}$
1	$F_1$	MULT1, SWITCH1 (then), DIVIS1	3	0.23
2	$F_2$	MULT3, SWITCH1 (else)	2	0.15
3	$F_6$	SWITCH2 (then), SYNC01, SWITCH3 (else)	3	0.23
4	$F_5$	F3, MULT5, SWITCH2 (else)	3	0.23
5	$F_3$	SYNC1, SWITCH3 (then)	2	0.15

Table 2

Test strategies and measurements values for case study 1

	$F_1$	$F_2$	$F_3$	$F_4$	$F_5$	$F_6$	$F_7$	GTD	$\Delta$
FTC	3	3	7	7	5	6	6		
Start-Small	X	X	X		X	X		24	2.7
Start-Big	X				X		X	14	5.1
Multi-Clue	X		X		X		X	21	1.5
All-Paths	X	X	X	X	X	X	X	37	1.5

incremental strategies, Start-Small and Start-Big offer different advantages between diagnosis effort and test difficulty. The expected size of indistinguishability set is close to 5 for Start-Big and close to 3 for Start-Small. This means that a detected fault is probably hidden in a set of 5 indistinguishable components for Start-Big while the set contains only 3 components for Start-Small.

### 6.2. Case study 2

The second diagram is presented in Fig. 8. Thirteen information flows are obtained from the SAO diagram. When needed, the input variable is added to make the difference between the branch of the SWITCH 1.1 exercised branch:

$$F_1 = \{M1, M3, M5, M6, M7\} | \text{OUT1},$$

$$F_2 = \{v\_4, M2, M3, M5, M6, M7\} | \text{OUT1},$$

$$F_3 = \{v\_5, M2, M3, M5, M6, M7\} | \text{OUT1},$$

$$F_4 = \{M1, M4, M5, M6, M7\} | \text{OUT1},$$

$$F_5 = \{v\_4, M2, M4, M5, M6, M7\} | \text{OUT1},$$

$$F_6 = \{v\_5, M2, M4, M5, M6, M7\} | \text{OUT1},$$

$$F_7 = \{M1, M8\} | \text{OUT1},$$

$$F_8 = \{v\_4, M2, M8\} | \text{OUT1},$$

$$F_9 = \{v\_5, M2, M8\} | \text{OUT1},$$

$$F_{10} = \{M3, M5, M8\} | \text{OUT1},$$

$$F_{11} = \{M4, M5, M8\} | \text{OUT1},$$

$$F_{12} = \{M3, M5\} | \text{OUT2},$$

$$F_{13} = \{M4, M5\} | \text{OUT2}.$$

The global test difficulty and the global diagnosability values with respect to the test strategies are shown in Table 3. It can be noted that although the global test difficulties are close to one another, the diagnosability value for Start-Big is much higher. It means that the expected diagnosis effort and preciseness are significantly more important when using Start-Big than Start-Small: the mean cardinality of the indistinguishability set in which a fault can be hidden is in the interval [6, 7].

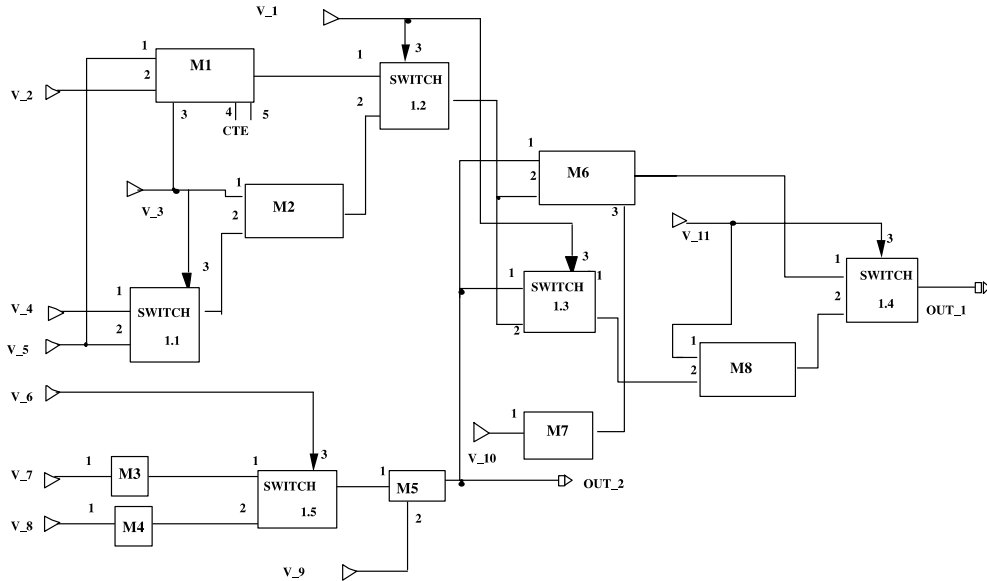


Fig. 8. Case study 2: SAO diagram.

Table 3  
Test strategies and measurements values for case study 2

	$F'_1$	$F'_2$	$F'_3$	$F'_4$	$F'_5$	$F'_6$	$F'_7$	$F'_8$	$F'_9$	$F'_{10}$	$F'_{11}$	$F'_{12}$	$F'_{13}$	GTD	$\Delta$
FTC	8	9	9	8	9	9	4	5	5	6	6	3	3	34	2.7
Start-Small	X						X	X	X		X	X	X	28	6.4
Start-Big		X				X	X	X		X				41	1.9
Mult-Clue	X		X			X	X	X		X				84	1.9
All-Paths	X	X	X	X	X	X	X	X	X	X	X	X	X		

6.3. Global case study

The global case study is the design obtained by connecting the PIL102 output from case study 1 to the V\_10 input to case study 2. 39 flows are to be exercised: 30 resulting from the combination of the flows  $F_3-F_7$  (case study 1) with the flows  $F'_1-F'_6$  (case study 2), the others remaining unchanged by concatenation. Fig. 9 shows the main results. We do not present the results of the All-Paths strategy which GTD value is too high to be interesting (=418). Fig. 9 highlights a comparison of the three designs: areas delimit each design while arrows show the evolution of values when concatenating design 1 and 2.

An ordering appears in terms of test costs: design 1 is easier to test than design 2 and design 3 is the hardest to test. However, in terms of diagnosability, the trend depends on the selected strategy. First, note that the GDA2 axiom is verified on this example: the diagnosability of design 3 for Mutiple-Clue is intermediate between the diagnosability values of design 1 and 2. Besides, in the case of Start-Big there is a very important degradation of diagnosability after concatenation. For the global design, the diagnosability is near 11. As far as Start-Big is concerned, another outcome of this

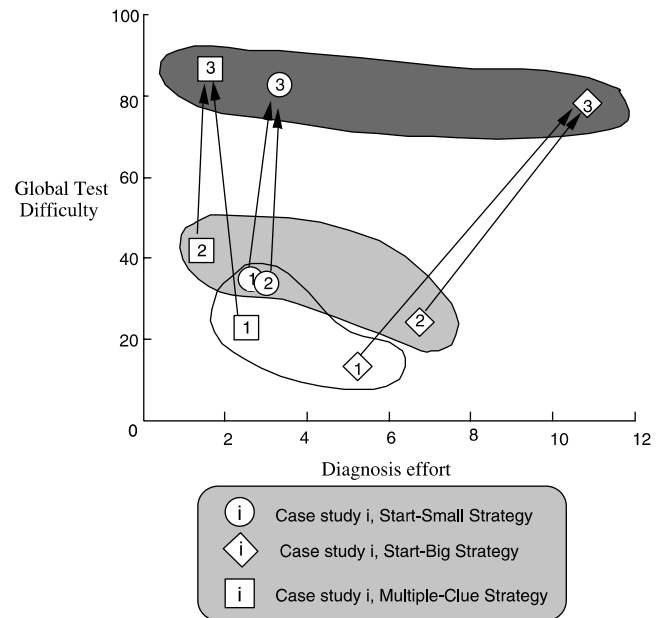


Fig. 9. Global case study: SAO diagram.

case study is that testing and diagnosing separate designs require a lower effort than carrying out the same

operations on the composed design. Indeed, one note that both the global test difficulty and the diagnosability measures of design 3 are greater than the respective metric values for designs 1 and 2. Start-Small behaviour is good since the test difficulty is of the same order than for Start-Big while the diagnosability increase moderately.

Thus, the global diagnosability measure for data flow designs offers an easy way of comparing designs, as well as a method for appraising test strategies efficiency in terms of diagnosis effort and preciseness. So, diagnosability measures provide an estimate of the design quality. This estimate allows one to take into account the testing context to guide the designer when allocating testing resources.

## 7. Reusing diagnosability axiomatization for design by contract

In this section, we address the question of measuring the impact of assertions (or contracts in a design by contract OO approach) on system diagnosability. The objective of this study is to illustrate the reuse of the already given axiomatization for data-flow designs to this specific question (a complete study of design by contract impact on another quality factor, namely *robustness*, can be found in (Baudry et al., 2001)).

### 7.1. The problem domain: design-by-contract (or assertions)

The notion of software contract has been defined to capture mutual obligations and benefits among classes. Experience tells us that simply spelling out unambiguously these contracts is a worthwhile design approach (Jézéquel and Meyer, 1997; Jézéquel et al., 1999), that Meyer cornered the design by *contract* approach to software construction (Meyer, 1992). This design by contract approach has a sound theoretical basis in relation to partial functions, and provides a methodological guideline for building robust, yet modular and simple systems. The design by contract approach prompts developers to specify precisely every consistency condition that could go wrong, and to explicitly assign the responsibility of its enforcement to either the routine caller (the client) or the routine implementation (the contractor). Along the line of abstract data type theory, a common way of specifying software contracts is to use boolean assertions called pre- and post-conditions for each service offered, as well as class invariants for defining general consistency properties. A contract carries mutual obligations and benefits: the client should only call a contractor routine in a state where the class invariant and the precondition of the routine are verified. In return, the contractor promises that when the

routine returns, the work specified in the postcondition will be done, and the class invariant is still respected.

A failure to meet the contract terms indicates the presence of a fault, or bug. A precondition violation points out a contract broken by the client: the contractor does not then have to try to comply with its part of the contract, but may signal the fault by raising an exception. A postcondition violation indicates a bug in the routine implementation, which does not fulfill its obligations. When a contract is violated, it indicates the part of the code where a wrong program state has been detected. With no contract embedded in the software, the failure would have been detected elsewhere, perhaps at the system output. Since the scope of diagnosis (the number of statement in which the fault must be located) is reduced when contracts catch the presence of a fault, it can be seen that contracts help the diagnosis task. The question is: to what degree do contracts reduce the diagnosis effort depending on their “strength” and number?

As presented in Fig. 10, a design-by-contract software should enable early detection of faults (during their

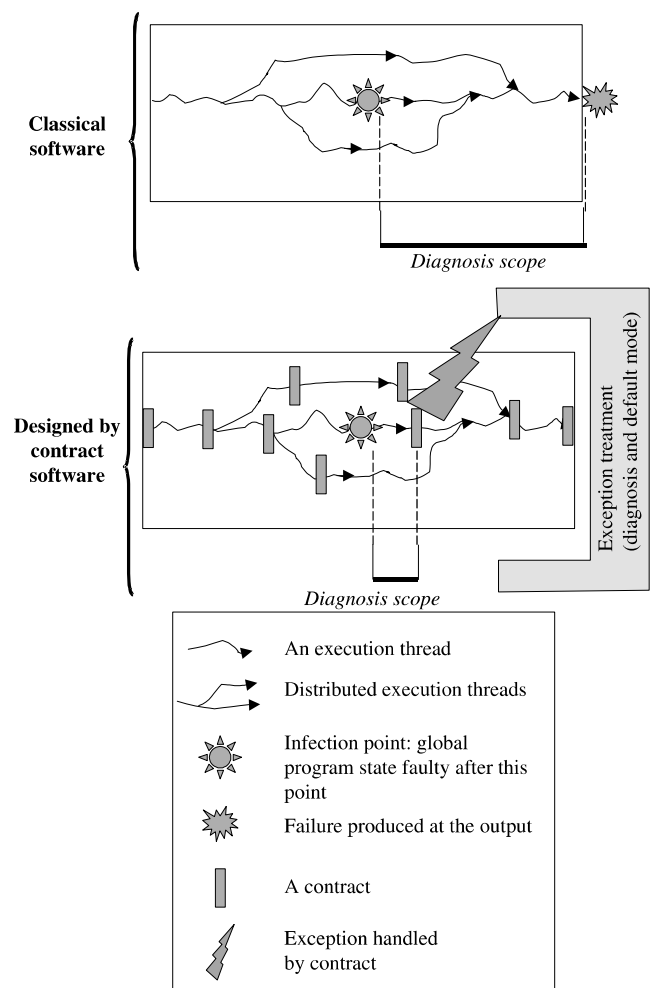


Fig. 10. Contracts for early detection of a fault.

propagation to the outputs) and help locating the faulty part of the software by reducing the “diagnosis scope” in which a fault must be located. What is of great interest with contracts is the fact that their efficiency is not dependent on possibly distributed execution of the software. While faults are really difficult to locate in a distributed execution environment, using contracts there is a good probability of pinpointing where the fault occurred.

Our entry points to diagnosability measurement are the statements executed when a failure occurs or when a contract (or assertion) detects the fault. The notion of component is forgotten here, since we concentrate first on a particular software execution.

Assumptions/propositions on diagnosis in an assertion/contract based programming context:

1. The software is assumed to be faulty: there exists an execution of the system that would provoke a failure if it were not detected by a contract.
2. Contracts are assumed to be correct.
3. A fault can be modeled as an invalid state in the global program state, which differs from the expected one after the execution of a statement. We call this statement the faulty statement, even if it is not necessarily the cause of the failure (that for example can be an omitted statement).
4. The main diagnosis task consists of locating this divergence point.
5. As a consequence, if an execution flow is faulty on multiple points, the diagnosis will point out the first divergence point (faults that compensate each other are considered as negligible for a global estimate).

### 7.1.1. Definitions

**Definition (Execution flow).** An execution flow is a partially ordered set of statements and contracts (or assertions) that is executed by a given program. Implicitly, we only consider flows that would provoke a failure.

The statements are partially ordered because—and particularly in an OO system—some of the execution may be distributed on several threads. To simplify the mathematical model, we only consider a non-distributed flow. The exhaustive mathematical modeling of such flows is not presented because it does not significantly modify the results of the measurements while it makes the model much more complex. In our case, an execution flow is equivalent to a dynamic slice of the system.

**Definition (Indistinguishable statements).** Two statements are indistinguishable from each other if they are bounded by consecutive contracts in an execution flow (or the entrance and output of the flow).

**Definition (Indistinguishability set).** An indistinguishability set corresponds to a set of indistinguishable statements.

**Definition (Size of an indistinguishability set).** The size of an indistinguishability set is equal to the cardinality of this set. The smallest size is equal to 1; it is associated with the case when the suspected set is made up of only one statement.

For local measurements (attached respectively to a statement and to an execution flow), the diagnosability is more directly expressed in terms of diagnosis effort (in that case, a diagnosability improvement corresponds to a reduced diagnosis effort) while the global diagnosability measure (attached to a system) is related to diagnosis preciseness.

**Definition (Local diagnosis effort ( $\delta$ )).** The local diagnosability  $\delta$  of a statement Stat in an execution flow  $F$  is the probable effort needed for determining that Stat is faulty in  $F$  knowing the number of statements, the number and repartition of contracts and their efficiency.

**Definition (Local diagnosis effort for a flow ( $\delta_{\text{eff}}$ )).** The global diagnosis effort for a flow  $F$  is the probable effort needed for pointing out the faulty statement, knowing that a fault is detected in  $F$ , knowing the number of statements, the number of contracts and their efficiency.

**Definition (Global diagnosability of a system ( $\Delta$ )).** The global diagnosability of a system  $S$  is the probable degree of preciseness obtained depending on the embedded contracts density and quality.

From these measurement definitions, derived from the informal one, one can deduce the measure expected profiles. A diagnosis effort is a ratio extensive measure (operation such as addition are possible since it is a counting measure). Local diagnosability and diagnosis effort should thus express the probable size of an indistinguishability set. These measurements thus take their domain value into  $[1 \dots + \infty]$ . Note that a good diagnosability corresponds to a low diagnosis effort. For the global diagnosability  $\Delta$ , a difficulty comes from the fact that there is no general relationship between the size of the execution flow and the size or architecture of the system. What we want to measure is the degree of diagnosis precision obtained with a certain proportion/quality of contracts in the system, compared to the same system with no contracts (as an absolute reference value).  $\Delta$  is also a ratio measurement but it is intensive, in the sense that values can not be easily combined. For  $\Delta$  a good diagnosability corresponds to a 1 value and the worse one to 0 (when no contracts or assertions allow the reduction of the diagnosis scope).

### 7.1.2. Axiomatization

Measures profiles:

$\delta$  : Statement  $\times$  Flow  $\rightarrow$  Real over  $[1 \dots + \infty]$ ,

$\delta_{\text{eff}}$  : Statement  $\rightarrow$  Real over  $[1 \dots + \infty]$ ,

$\Delta$  : System  $\rightarrow$  Real over  $[0 \dots 1]$

#### 7.1.2.1. Local diagnosability axioms

**LDA1 (Statements comparison).** All statements in an execution flow are comparable in terms of local diagnosability. Two execution flows are also always comparable.

**LDA2 (Relationship between indistinguishable statements).** Indistinguishable statements have the same local diagnosis effort value.

**LDA3 (Flow with no contract (or assertion)).** A flow (and statements of the flow) with no contracts has a diagnosis effort equal to the number of statements of the flow.

**LDA4 (Relationship between local diagnosability and diagnosis effort).** The diagnosis effort for a flow is bounded by the upper and lower values of diagnosis efforts of its statements.

The impact of the system concatenation on the execution flow is not simply foreseeable: the corresponding axiom has thus no sense.

**LDA5 (Monotonicity for contract (assertion) addition).** In a flow, the local diagnosis effort values (for statements and for the flow) cannot increase by addition of a contract in the system.

**LDA6 (Monotonicity for contract improvement).** The improvement of a contract in a system must reduce the diagnosis effort of the flow (and of statements upward the contract execution) if this contract is involved in the flow.

#### 7.1.2.2. Global diagnosis axioms

**GDA1 (System comparison).** Two systems are always comparable in terms of global diagnosability.

**GDA2 (System with no contract (or assertion)).** A system with no contract has a diagnosability value of 0 (i.e. preciseness degree).

**GDA3 (Monotonicity for system concatenation).** The global diagnosability of a system obtained by concatenation of two systems  $S_1$  and  $S_2$  is bounded by the lower and upper diagnosability values of  $S_1$  and  $S_2$ .

**GDA4 (Monotonicity for contract (assertion) addition).** The global diagnosability of a system cannot decrease by addition of a contract in the system.

**GRA5 (Monotonicity for contract improvement).** The improvement of a contract of any component  $C_i$  in a system must increase its global diagnosability.

**7.1.2.3. Inter-level axioms.** The inter-level axioms cannot easily be foreseen, the relationships between a flow and a system being unclear. All possible execution flow should be known to induce the global degree of diagnosis preciseness.

**IDA1 (Inter-level relationship).** An injective function relates all local diagnosis effort values to global diagnosability.

## 7.2. Diagnosability measurements

Since this section describes the adaptation of the axiomatization to another software paradigm, we briefly present the measurement and report results. The diagnosability measurement is based on the probabilities for each contract to detect a fault, i.e. their efficiency.  $\text{Det}_i^j$  ( $i \in [1 \dots n\_contracts]$  and  $j \in [i \dots n\_contracts]$ ) is the probability that a faulty statement of  $IS_i$  has to be detected by the contract  $j$  knowing that no intermediary contract has detected this faulty state. To model the fact that the closer a contract is to the faulty statement  $i$  the more probable it can detect the fault (i.e.  $pk$  decreases when  $k$  grows,  $k \in [i \dots n\_contracts]$ ), we consider that: with contracts with a  $p$  uniform value of efficiency, if the first executed contract has a probability  $p$  to detect the faulty state, the second has only  $\alpha p$  probability to detect it, the third one  $\alpha^2 p$  and so on.  $\alpha$  is a constant absorption coefficient ( $\alpha \in [0 \dots 1]$ ). If  $\alpha$  is equal to 0, it means that only the first executed contract can detect the fault. The local diagnosability and the global ones can be deduced in function of the  $\text{Det}_i^j$  knowing that any statement may be faulty. Indeed, if the fault is detected by contract  $j$ , then the size of the diagnosis scope is equal to the number of statements between the faulty one to the contracts detection.

The number and distribution of contracts, their efficiency and the absorption coefficient are the main parameters of the model. In any system, an appropriate instrumentation of the code would lead to an exact counting for  $N_{\text{stat}}$  and  $n\_contracts$  for a given flow. Here we will make these parameters vary to study their impact. To fix them we used a mutation analysis (Offutt et al., 1996; Baudry et al., 2000): it shows that contracts efficiency varies between 0.17 for less efficient ones to 1 for the best one. An average for a reasonable effort is around 0.7. For contracts with efficiency of 0.4, the number of probable traversed contracts before detection

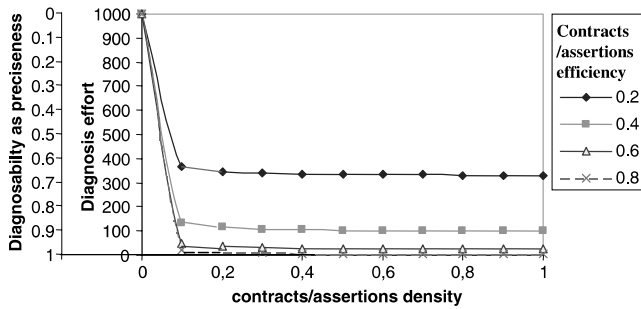


Fig. 11.  $\delta_{\text{eff}}$  and  $\Delta$  results—main results.

is comprised between 3.1 and 6.1 for 10 maximum contracts. A significant loss of precision for the global diagnosability measurement can be due to a bad estimation of the absorption coefficient (particularly if contract efficiency is between 0.2 and 0.7). However the differences remain acceptable and will never contradict the axioms. We used the Pylon library (<http://www.eiffel-forum.org/archive/arnaud/pylon.htm>) as a case study. It is a small, portable, freely available Eiffel library for data structures and other basic features. The class diagram is composed of 50 classes and 134 relations (inheritance, associations, etc.). In our case, experiments with mutation analysis give a range of values for absorption coefficient around 0.8 for the pylon Eiffel library.

Fig. 11 presents the same results with both scales, in terms of “absolute” diagnosis effort and in terms of relative preciseness. Diagnosis effort curves are given for various contracts efficiency with a 1000 statements flow—the absorption coefficient is equal here to 0.8.

The global diagnosability is directly obtained by any flow diagnosis effort curves, and for example by those given in Fig. 8. The gap between local diagnosis effort values and global diagnosability is thus bridged thanks to a good property of the model: diagnosability of a system only depends on its contracts efficiency and density as shown on Fig. 11.

### 7.3. Results and conclusions

Let us interpret the results of the Fig. 11. First, we remark that the introduction of contracts quickly enhances the global diagnosability of the system. Second, the addition of many contracts does not (high contract density) significantly improve a system global diagnosability (that is upper bounded around 0.6 with 0.2 efficient contracts or around 0.9 with 0.4 efficient contracts). Third, the quality of the contracts is more important than their number, since it is the only way to make the upper bound to diagnosability increase.

The conclusions we can deduce from this measurement are the following:

- A 0.2 contract/assertion density is enough to reach the upper bound of diagnosability for a given contract average efficiency. It has to be noted that in good OO designs the size of methods is often small, and includes a small number of statements. A 0.2 contract density corresponds to a good and possible density for an OO system. In most cases, the use of assertions in the body of methods is thus useless. This result could not be easily predicted without a mathematical model.
- Quality is better than quantity. For a same density, the quality of contracts highly changes the diagnosability. It is better to put the effort on a good design with high encapsulation and well defined interfaces (supporting clearer properties derivable into contracts) than too put the effort on defensive assertions, that often are unclear and dependent from the code.

## 8. Conclusion

It is a non-sense to allow an easy fault-detection by improving the fault-revealing trend of a software design if no way of facilitating the fault location is provided. A quality predictor of testability should not be used without a complementary diagnosability estimate. The proposed integrated approach for testability/diagnosability provides this combined view of the validation difficulties.

This approach to diagnosability is first dedicated to data flow software and a large part of the paper focused on a formal model adapted to this application context. The proposed diagnosability measurement is consistent with the stated axioms and the data-flow context. Trade-offs between global test difficulty and diagnosis preciseness have been commented on a piece of data-flow software design. A very interesting point is the chosen measurement methodology based on properties, i.e. the axioms: axioms are kind of measurement requirements at the frontier between the intuition and the effective and dedicated measure. A dedicated measure is thus similar to the implementation of the measurement requirements seen as a specification. At a high-level specification, axioms can be reused/adapted/completed to other application contexts, as it has been illustrated by adapting the axiomatization to obtain a dedicated measure of diagnosability for the design by contract application context.

## Acknowledgement

The authors are grateful to Jean-Marc Jézéquel for his help and advices to improve the quality of the paper concerning design by contract.

## Appendix A. Indistinguishable components

Let  $D$  be a design including  $n$  components  $C_i$ , and let  $f_i$  denote a flow in  $D$

$$\text{Is\_Exercized}(C_i, f_j) = \text{true} \quad \text{if } (C_i \in f_j),$$

$$\text{Is\_Exercized}(C_i, f_j) = \text{false} \quad \text{if } (C_i \notin f_j).$$

*Case of cross-checking strategies*

Let CS be a cross-checking test strategy;  $\text{CS} = \{f_j, j \in [1 \dots p]\}$ ,

$C_i$  et  $C_k$  are indistinguishable for CS iff:

$$\begin{aligned} C_i \overset{\text{CS}}{\sim} C_k &= (\forall j \in [1 \dots p], \\ &(\text{Is\_Exercized}(C_i, f_j) \wedge \text{Is\_Exercized}(C_k, f_j)) \vee \\ &(\neg \text{Is\_Exercized}(C_i, f_j) \wedge \neg \text{Is\_Exercized}(C_k, f_j))). \end{aligned}$$

*Case of incremental strategies*

Let IS be an incremental test strategy;  $\text{IS} = (f_j, j \in [1 \dots p])$  (IS is an ordered list of flows),

$C_i$  et  $C_k$  are indistinguishable for SI iff:

$$\begin{aligned} C_i \overset{\text{IS}}{\sim} C_k &= (\text{Is\_Exercized}(C_i, f_1) \wedge \text{Is\_Exercized}(C_k, f_1)) \\ &\vee (\exists j \in [2 \dots p], \\ &(\text{Is\_Exercized}(C_i, f_j) \wedge \text{Is\_Exercized}(C_k, f_j)) \wedge \\ &\forall l \in [1 \dots j - 1] \\ &(\neg \text{Is\_Exercized}(C_i, f_l) \wedge \neg \text{Is\_Exercized}(C_k, f_l))). \end{aligned}$$

## Appendix B. Axiomatization

*B.1. Local diagnosability axioms:*

**LDA1 (Component comparison).** All components of a design are comparable in terms of local diagnosability.

Let  $S$  be a test strategy,  $D$  a design composed of  $n$  components  $C_i$  ( $i \in [1 \dots n]$ ). A total order relationship  $\leq_{\delta, S}$  exists between components such as:

$$\forall i, j \in [1 \dots n], \quad (C_i \leq_{\delta, S} C_j) \iff (\delta(C_i, D, S) \geq \delta(C_j, D, S)),$$

$(C_i \leq_{\delta, S} C_j)$  means that  $C_j$  has a better diagnosability than  $C_i$ , i.e. it implies a lower diagnosis effort.

**LDA2 (Relationship between indistinguishable components).** Indistinguishable components have the same local diagnosability.

Let  $S$  be a test strategy,  $D$  a design composed of  $n$  components  $C_i$ , ( $i \in [1 \dots n]$ ). Let IS be an indiscernability set of  $D$  for  $S$ :

$$\forall i, j \in [1 \dots n], \quad (C_i \in \text{IS}) \wedge (C_j \in \text{IS}) \Rightarrow (C_i \overset{\delta, S}{=} C_j).$$

Indeed, two indistinguishable components are both suspected in terms of diagnosis. They are linked to each other in the fault localization process, and the effort needed to determine which of the indistinguishable component is faulty is the same. So, all the components of an indistinguishability set have the same local diagnosability.

**LDA3 (Design concatenation).** The local diagnosability of a component cannot increase when concatenating this design to another one.

Let  $D_1$  et  $D_2$  be two designs,  $C_1$  a component of  $D_1$  and  $C_2$  a component of  $D_2$ :

$$\delta(C_1, \text{Concatenate}(D_1, D_2), S) \geq \delta(C_1, D_1, S),$$

$$\delta(C_2, \text{Concatenate}(D_1, D_2), S) \geq \delta(C_2, D_2, S).$$

Indeed, the indistinguishable components in  $D_1$  remain indistinguishable after the concatenation of  $D_2$  to  $D_1$ . Moreover, some components of  $D_1$  may be indistinguishable from some components of  $D_2$  after concatenation. Indistinguishability set sizes can only increase when concatenating designs.

**LDA4 (Component instantiation).** In a design, the diagnosability of a component cannot increase by instantiation of another component.

Let  $D$  be a design,  $S$  a test strategy,  $C_1$  and  $C_2$  two different components of  $D$ :

$$\delta(C_2, \text{Instance}(D, C_1), S) \geq \delta(C_2, D, S).$$

**LDA5 (Addition of an observation point).** The diagnosability of a component can only increase by addition of an observation point to the enclosing design (i.e. the diagnosis effort is lower).

Let  $D_1$  be a design,  $C$  a component of  $D_1$ :

$$\delta(C, \text{Add\_output}(D_1), S) \leq \delta(C, D_1, S).$$

Some components which are indistinguishable without the addition of an observation point, can be isolated afterwards. So, the diagnosis effort cannot increase and component diagnosability should only increase.

*B.2. Global diagnosability axioms:*

**GDA1 (Design comparison).** Two designs are always comparable in terms of diagnosability.

Let  $S$  be a test strategy,  $D_1$  and  $D_2$  two designs. A total order relationship exists  $\leq_{\Delta, S}$  such as:

$$\left( D_1 \leq_{\Delta, S} D_2 \right) \iff (\Delta(D_1, S) \geq \Delta(D_2, S)),$$

$(D_1 \leq_{\Delta, S} D_2)$  means that  $D_2$  a higher diagnosability than  $D_1$ , i.e.  $D_2$  implies a lower diagnosis effort than  $D_1$ .

**GDA2 (Design concatenation).** The diagnosability of a design obtained by concatenation of two designs  $D_1$  and  $D_2$  is lower than the diagnosability of the most diagnosable design among  $D_1$  and  $D_2$ .

Let  $S$  be a test strategy,  $D_1$  and  $D_2$  two designs,

if  $D_1 \underset{\Delta, S}{\geq} D_2$  then  $\text{Concatenate}(D_1, D_2) \underset{\Delta, S}{\leq} D_1$

else  $\text{Concatenate}(D_1, D_2) \underset{\Delta, S}{\leq} D_2$ .

The resulting diagnosability cannot be easily predicted:  $\Delta(D, S)$  may be in between  $\Delta(D_1, S)$  and  $\Delta(D_2, S)$ , or it may be lower than both  $\Delta(D_1, S)$  and  $\Delta(D_2, S)$ .

The behaviour of global diagnosability with respect to the Instantiate operation cannot be axiomatized, since it is a major operation which deeply modifies the process attribute (test strategy). However, the addition of an observation point is a minor operation and it can be axiomatized.

**GDA3 (Observation point addition).** For any design, its global diagnosability cannot decrease by addition of an observation point.

Let  $D_1$  and  $D_2$  be two designs,  $S$  a test strategy:

$\Delta(\text{Add\_output}(D_1, S) \leq \Delta(D_1, S))$ .

In the worst case, the addition of an observation point will not change the diagnosability of internal components. If this operation modifies some components diagnosability, it can only improve it: the global diagnosability should only be improved, it cannot decrease.

### B.3. Inter-level axioms

The knowledge of local diagnosability should be sufficient to deduce design global diagnosability.

**IDA1 (Inter-level relationship).** An injective function relates local diagnosability to global diagnosability.

## References

- Agrawal, H., Horgan, J., London, S., Wong, W., 1995. Fault localization using execution slices and dataflow tests. In: 6th International Symposium on Software Reliability Engineering, Toulouse, France, pp. 143–151.
- Al-Hayek, G., Le Traon, Y., Robach, C., 1997. Impact of system partitioning on test cost. IEEE Design & Test of Computers 14, 64–74.
- Baudry, B., Hanh, V.-L., Jézéquel, J.-M., Le Traon, Y., 2000. Testable components: yet another mutation-based approach. In: Proceedings of the Symposium on Mutation Testing for the New Century (Mutation'2K), San Jose, CA, pp. 69–76.
- Baudry, B., Le Traon, Y., Jézéquel, J.-M., 2001. Robustness and diagnosability of Designed by Contracts OO Systems. In: 7th International Software Metrics Symposium (Metrics 2001), London, England.
- Briand, L.C., Morasca, S., Basili, V.R., 1996. Property-based software engineering measurement. IEEE Transactions on Software Engineering 22, 68–86.
- Caspi, P., Halbwachs, N., Pilaud, D., Plaice, J., 1987. LUSTRE: a declarative language for programming synchronous systems. In: 14th Symposium on Principles of Programming Languages (POPL 87), Munich.
- Fenton, N.E., 1991. Software Metrics: A Rigorous Approach. Chapman & Hall, London, UK, p. 337.
- Jézéquel, J.-M., Meyer, B., 1997. Design by contract: the lessons of Ariane. Computer 30 (1), 129–130.
- Jézéquel, J.-M., Train, M., Mingins, C., 1999. Design-patterns and contracts. Addison-Wesley, October 1999. ISBN 0-201-30959-9.
- Kamkar, M., 1995. An overview and comparative classification of program slicing techniques. Systems Software 31, 197–214.
- Khalil, M., Le Traon, Y., Robach, C., 1998. Automated strategies for software diagnosis. In: Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE'98), Paderborn, Germany, November 1998.
- Korel, B., 1997. Computation of dynamic program slices for unstructured programs. IEEE Transactions on Software Engineering 23, 17–34.
- Le Traon, Y., 1997. Analyse conjointe logiciel-matériel de la testabilité de systèmes flots de données. PhD Thesis, Institut National Polytechnique de Grenoble, Grenoble, France.
- Le Traon, Y., Robach, C., 1997. Testability measurements for data flow designs. In: 4th International Software Metrics Symposium (Metrics'97), Albuquerque, New Mexico, pp. 91–98.
- Le Traon, Y., Ouabdesselam, F., Robach, C., 1998. Software diagnosability. In: Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE'98), Paderborn, Germany, November 1998.
- Le Traon, Y., Ouabdesselam, F., Robach, C., 2000. Analyzing testability on data-flow designs. In: Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'2K), San Jose, CA, pp. 162–173.
- Meyer, B., 1992. Applying “Design by contract”. IEEE Computer 25 (10), 40–52.
- Offutt, A.J., Pan, J., Tewary, K., Zhang, T., 1996. An experimental evaluation of data flow and mutation testing. Software Practice and Experience 26 (2).
- Ouabdesselam, F., Parissis, I., 1996. Specification-based testing of synchronous software. In: ACM-SIGSOFT 4th Symposium on the Foundations of Software Engineering, San Francisco, pp. 127–134.
- Robach, C., Wodey, P., 1989. Linking design and test tools: an implementation. IEEE Transactions on Industrial Electronics 36, 286–295.
- Ross, D.T., Schoman, K.E., 1977. Structured analysis for requirements definition. IEEE Transactions on Software Engineering SE-3, 6–15.
- Shepperd, M., Ince, D., 1993. Derivation and Validation of Software Metrics. Oxford University Press, Oxford, UK, p. 167.
- Voas, J.M., 1995. Software testability measurement for assertion placement and fault localization. In: 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG'95), Saint-Malo, France.
- Weiser, M., 1982. Programmers use slices when debugging. Communication of ACM 25, 446–452.
- Weiser, M., 1984. Program slicing. IEEE Transactions on Software Engineering 10, 352–357.

**Yves Le Traon** received the engineering degree, in 1994, and a Ph.D. in Computer Science, in 1997, from the Institut National Polytechnique de Grenoble, France. He is now an Assistant Professor at the University of Rennes I and a member of the IRISA research institute. His research fields are testing, object-orientation, design-for-testability and software measurement.

**Farid Ouabdesselam** is a professor at Joseph Fourier University, Grenoble, France. Previously, he has served on the faculties of UC Santa Barbara, and Queen's University, Kingston, Ontario. He has also been project leader at Bull. His current research interests include synchronous reactive software testing and verification, formal specification, and formal approaches to software quality.

**Chantal Robach** is a professor at the INPG, head of the Computer Department of the ESISAR-INPG school and she manages the "Validation of Integrated Systems" group in LCIS laboratory. She

graduated from the ENSIMAG Computer Engineering School, Grenoble, in 1973 and she got the "Docteur-es-Sciences" degree from the INPG in 1979. Her main research interests include the validation of hardware/software systems, hardware/software testability, system diagnosis, design of testable co-designed systems. She is a member of IEEE.

**Benoit Baudry** is a Ph.D. student in computer science in the IRISA laboratory at the University of Rennes I, France. His research interests concern the testability and the reliability of object-oriented and component-based systems.