Reducing Complexity of Data Flow Testing in the Verification of a IEC-62304 Flexible Workflow System

Federico Cruciani\textsuperscript{1} and Enrico Vicario\textsuperscript{2}

\textsuperscript{1} I+ s.r.l., Piazza Puccini, 26 - 50144 Florence, Italy,
\texttt{f.cruciani@i-ius.it}
\textsuperscript{2} Università degli Studi di Firenze

\textbf{Abstract.} In the development of SW applications, the workflow abstraction gives primary relevance to the way how some process can be accomplished through a sequence of connected steps. This largely conditions analysis, implementation architecture, and verification. In particular, testing activities are naturally oriented towards a data flow approach, which effectively exercises dependencies among steps. In several application scenarios, the workflow model cannot completely determine the sequencing of actions and it must rather leave space to variability. While easily encompassed both in the analysis and implementation stages, this comprises a major hurdle for the testing stage due to the explosion in the number of allowed execution orders and paths.

We address the problem reporting on the verification of the control software of a Computer Assisted Surgery system. In this case, the workflow abstraction captures the constraints of a medical protocol, and variability in the order of steps reflects dynamic adaptation of the course of actions to the specific characteristics of each patient. This largely increases the testing effort needed to accomplish the prescriptions of the IEC-62304 certification standard. To cope with the problem, we show how data flow analysis can be used to identify an appropriate set of constraints that can be exploited in the verification stage, so as to reduce the test suite while preserving coverage.

\textbf{Keywords:} Workflow architecture, Workflow verification, Data Flow testing.

1 Introduction

Workflow applications are commonly used to automate processes that require a sequence of steps to be performed in a certain order to complete a task\textsuperscript{[2]}. Typical areas of application include business process management, manufacturing and supply management, healthcare protocols. In general, a workflow application can be conveniently implemented by letting an executable specification of a process be enacted by an operational engine. This enables consistent reuse of the engine among multiple applications, facilitates evolutionary maintenance of processes, and centers the overall SW life cycle on a process-oriented model that can be effectively agreed among software engineers, stakeholders, and users. In so doing, the overall development can be accompanied by effective tools including UML Activity diagrams supporting representation\textsuperscript{[3]}\textsuperscript{[4]}, formal models providing theoretical foundation\textsuperscript{[5]}, and frameworks providing a basis for effective enactment\textsuperscript{[6]}.

Various testing strategies for workflow applications have been proposed, mainly relying on the automated derivation of a Control Flow Graph (CFG) abstraction\textsuperscript{[6]}\textsuperscript{[7]} that drives test case selection or coverage analysis. CFGs can be derived directly from workflow specifications such as UML Activity diagrams\textsuperscript{[4]}\textsuperscript{[3]}. A data flow model can be built from the requirements and this model can conveniently be exploited also in formal verification, following a kind of model-based
approach to design develop, and test the system. Model verification of workflows allows indeed the early detection of sequencing problems such as deadlock or non-terminating behaviors [8].

In the transition stage of the SW life cycle, and in particular in the acceptance step, the process model provides a native abstraction enabling application of the consolidated theory of data flow testing in functional perspective [9]. In particular, according to all-uses criterion, the workflow is operated so as to exercise at least one path from each step where some relevant variable is modified to each the next step where the same variable is used. Though not prescribed by certification standards, this criterion has proven to effectively increase fault coverage capability with respect to branch coverage [10] while maintaining the testing effort in the range of polynomial complexity.

However, when testing comes to input generation and test execution, full coverage of the all-uses criterion still remains a complex task, especially when execution involves user interfaces and physical system devices. The complexity is further exacerbated whenever the automated process is not completely determined and rather enables multiple orders of execution determined during the run-time, resulting in a so-called flexible workflow [11] [12]. This case is relevant for any procedure where some of prescribed actions can be executed in different orders without compromising the integrity of the overall process. In the end, this kind of flexibility often serves to smooth the stiff mechanism of workflow applications, which work finely in setting rules more than accommodating exceptions. This takes a specific relevance in the healthcare context, where a crucial role is played by dynamic adaption of the execution order of a protocol according to the course of actions applied to a specific patient.

In this paper, we report on testing activities performed in the verification of the control software of a Computer Assisted Surgery system subject to ISO-62304 standard for medical software [1]. The SW under test is organized as a flexible workflow application that accompanies the steps of a surgical protocol. During the certification process, in order to compensate lacks in the structural coverage of component items, integration testing was planned and executed so as to attain all-uses coverage of the workflow specification. In doing so, the complexity of the test plan could yet be limited through the assumption of design choices that statically guarantee equivalence conditions among different paths and thus permit a substantial simplification of the test suite while preserving its coverage capability.

The rest of the paper is organized as follows. The characteristics of the application case and its testing requirements are described in Sect. 2. In section 3, we introduce a model abstracting data flow behavior of a workflow in a high-level Directed Acyclic Graph (DAG) showing how this can be extracted from a specification accepting multiple execution orders, and how this reduces testing complexity in the application case. Conclusions are drawn in Sect. 4.

2 Testing requirements for a Computer Assisted Surgery system

Miño³ is a workflow application for Computer Assisted (image guided) Surgery applied to knee arthroplasty. The software is based on BLU-IGS, an ad-hoc workflow engine optimized for orthopaedic surgical procedures that supports a product line of softwares for hip arthroplasty, kinematic analysis, knee prosthesis implant and revisioning.

Miño integrates an infrared optical localizer of surgical tools, used to track position of fiducial markers placed to the patient’s bone, so as to provide a reference system through which the anatomy of the patient limb is then reconstructed by means of registration of anatomical points through a pointer tool. Data acquired through the pointer are then used to build the anatomical reference systems for tibia and femur allowing an intra-operative planning of the final prosthesis.

³ Miño is developed by I+ s.r.l. as part of the BLU-IGS system distributed by Orthokey LLC: http://www.orthokey.com/index.php/totalknee.
implant. Surgical tools are then referred with respect to these anatomical reference systems and navigated during the operation to verify the correct positioning of the tibial and femoral components.

The control software accompanies the surgeon along a workflow protocol composed by phases or steps including tibial and femoral registration, implant positioning planning, navigation of femoral tibial cut. Many of these phases determine choices that condition the subsequent steps (e.g. the selection of the type of prosthesis to implant). The process is composed by data acquisition and operational steps strictly bound to the conventional surgical protocol. As in most existing competitors products, the current version of Miró follows a fixed execution order, forcing the surgeon to adopt his way of performing the operation to the built-in procedure. This lack of flexibility now appears to be a major hurdle for the adoption of Computer Assisted Surgery, despite this has been demonstrated to improve the quality of results, in particular in restoring the neutral alignment of the leg[13][14][15]. To cope with this issue, the new BLU-IGS version is moving towards a more flexible workflow engine, allowing the surgeon to vary the execution order, either to best fit his way of operating, or according to the patient anatomy. Some of the phases in the procedure are constraining, e.g. the type of prosthesis to implant. In the fixed workflow solution, this choice has to be performed at the initial stage of the operation, thus determining the behavior of some of the subsequent steps. Flexibility in this case would allow to postpone constraining choices to a latter phase of the operation, when the anatomy of the patient is more clear and the choice can leverage on a more complete understanding of the case.

2.1 Testing requirements for IEC 62304

The international standard IEC 62304 specifies life cycle requirements for the development of medical software and software within medical devices [1]. A Crucial element of the standard is the concept of design for patient safety, for which a fundamental role is played by risk analysis (including hazard identification), evaluation, and control.

In the case of Miró, the attainment of this objective had to face the infamous (yet common in real practice) problem of components classified as Software Of Unknown Provenance (SOUP). Specifically in this case, these are software items integrated in the overall 20000 lines of C++ code of Miró, that had been already developed and generally available but that had not been developed for the purpose of being incorporated into a medical device subject to certification requirements. In order to compensate the presence of SOUP components with non-compliant structural coverage
of unit tests, a higher responsibility and effort was charged on integration testing. Specifically, a grey-box testing approach was planned (and agreed in the certification process), with the goal of covering all the interactions among components for which unit testing coverage had not been attained. To this end, the test plan was targeted to attain def-use coverage (for each definition of any variable of global scope, cover at least one path reaching each subsequent use of that variable) and all du-paths (for each definition of any variable of global scope, cover each path reaching each subsequent use of that variable) [9].

Disciplined reasoning on UML activity diagrams of the basic process identifies 3 paths that are sufficient to cover the def-use criterion and 9 sufficient to ensure all-du paths. However, the addition of flexible choices to the basic process dramatically increases the set of feasible executions and the set of combinations among definitions and uses of data coupling different steps of the protocol. In principle, since the process consists of 13 steps, this results in 13! possible orders of execution. Fortunately enough, the protocol includes dependencies between states that constrain feasible executions (e.g., the navigation of tibial cutting guide can not be executed before the registration of tibial reference system). Despite the reduction, this still results in 96 test cases, which subtends a huge testing and documentation effort due to at least two major factor exacerbating complexity. On the one hand, functional tests on the integrated system must be manually performed by simulating a complete intervention for each path, with a significant effort also in the generation of input data for each path. On the other hand, due to accuracy requirements on measurements taken by the system during the surgical protocol, the oracle verdict on the results of each single test requires that device measurements and numerical processing be shown accurate up to 1mm and 1° to get benefits compared to conventional procedure.

This type of complexities suggested that the code be partially refactored so as to implement a few basic design-for-testability principles that could permit a significant reduction in the complexity of the Test Plan. In this particular scenario, each variable, either a cutting plan, an anatomic reference system or a point acquired during registration, is bound to be defined only in a single step where its value is acquired or calculated. Its value will then be used by some other subsequent steps, but, the only portion of code where the value can be modified remains the step where the value is assigned. We are interested in testing the IUT with all-uses coverage [9] of the behavior model specified in functional requirements. In so doing, we guarantee that behaviors allowed by the functional specification are tested so as to cover all-nodes, all-edges and also a relevant subset of all paths. Specifically, paths are covered according so as to guarantee that for each variable $x$ defined in some step $X$ and later used in some step $Y$ without any intermediate side-effect on $x$, at least one test is performed that reaches $Y$ from $X$ without ever modifying $x$. We will show that in this case, the data flow testing applied in functional perspective, not only detects an effective set of paths to test, but also helps, adding some appropriate constraints, to keep under control the variability introduced by the execution of the process in a flexible workflow.

3 Abstraction and problem formulation

It is worth in this context, to exactly define and classify the type of workflow we are working on. From a data flow perspective each variable has an only point in which is defined and this can be verified by static inspection of the code.

There are two kinds of variables involved: some variables are used to calculate or measure quantitative data, and some variables that correspond to choices, for instance the type of implant the surgeon decide to use. This second set of variables, in other words, affects a subset of steps where based on their values one of several branches is chosen within the execution of the step itself, i.e. some steps implement different behaviors according to some pre determined condition.
As for the variables we have thus two kind of steps: some steps can be viewed as a basic block while some other steps hide execution branches. However, we can consider each step in the procedure as an extended basic block, i.e. a sequence of consecutive instructions always executed from start to finish that may contain branches.

We also assume that a set of dependency between states can be derived by static analysis of the process.

The surgical process is composed by a set of steps $S$ that must all be performed but can be serialized in any way satisfying a given partial order $\prec \subseteq S \times S$ reflecting the constraints of the surgical protocol.

Our workflow can thus be defined as $W = (S, \prec)$ where $S$ is the set of possible phases that can be executed in different combination in the process, and $\prec$ is a set of high-level dependencies among steps.

As an example, be $S$ the workflow composed by the set of steps $S = \{A, B, C, D, E, F, G\}$. The control flow across nodes is conditioned by a set of variables $V = \{a, b, c, d, e, f, g\}$ with global scope, i.e. variables that are defined and used in different activities.

![Diagram of workflow](image)

**Fig. 2.** Two possible execution order of the process.

The dependency between steps implies that some process constraints have to be introduced to avoid sequencing problems i.e. the use in a particular step of a variable that has not yet been assigned.

### 3.1 Data flow perspective

In order to define the set of constraints we need to analyze all steps in the workflow model specification. We need order constraints[16][12] to avoid the execution of a step $N$ which uses a variable that is defined in step $M$ prior to the execution of $M$. This kind of constraints indicates that two steps have to be executed in an exact order but could have other independent workflow steps in between. This dependency relation between two steps can be naturally defined by using the data flow perspective.

As in [9], it is here convenient to distinguish among c-uses and p-uses, i.e. references to variable that condition the value assigned to some other variable (computational use) or the the result of a decision determining the flow of actions in the control flow graph (predicate use), respectively. In particular, in our setting, a p-use can determine the choice among different ways how some activity is performed. For instance, the selection of a type of prosthesis in some activity $A$ might define a global variable $a$, which in turn is later p-used in some activity $F$ to select the way how some measure is taken. This results in different modes of execution for $F$, say $\{F_1, F_2, F_3, \ldots\}$, which may define different variables, say $\{f_1, f_2, f_3, \ldots\}$. It may also happen that in some subsequent activity $G$, the same variable $a$ is p-used again to select among different modes $\{G_1, G_2, G_3, \ldots\}$ each of which uses in respective manner the variables $\{f_1, f_2, f_3, \ldots\}$ defined in $F$. 

![Diagram of data flow](image)
This setting directly reflects explicit needs of the context of use, and it is quite easily implemented in a workflow oriented SW architecture. However, a major hurdle for its practical realization arises in the testing stage. In fact, attaining all-def coverage for this variety of behaviors is by far beyond the limits of a feasibility. And, relaxation of the aim from all-uses to all-edges does not substantially change the nature of the problem. In fact, tests are here performed at the system level, and each of them requires manual application of a sequence of physical steps, which basically reproduce those of a surgical operation. Just to give an idea of the order of complexity that can be reasonably afforded, in the certification of the first release of the product which did not include workflow flexibilities, the test suite specified in the test plan was made by 9 cases.

In order to complete the specification of this flexible workflow we need to introduce some definitions.

**Definition 1.** Let $M, N$ be two steps of the procedure and $adu(M), adu(N)$ the corresponding sets of all definitions and uses of variables. We will say that $N$ depends on $M$ if $\exists$ any variable $x$ such that $def(x) \in adu(M)$ and $p - use(x)c - use(x) \in adu(N)$.

**Definition 2.** Let $M, N$ be two steps and $adu(M), adu(N)$ the corresponding sets of definitions and uses of variables and $V(N)$ and $V(M)$ the corresponding set of used or defined variables, we will say that $N$ and $M$ are independent if $V(M) \cap V(N) = \emptyset$.

**Definition 3.** A Dependency Graph is a pair $DG = (S, \Gamma)$ where $S$ is a set of vertices each of them representing a step, and $\Gamma$ is a set of pairs $(x, y) \in S^2$ called set of directed edges between two vertices each of them representing a dependency between the two steps.

Looking at this problem in data flow perspective we can define for each possible state which variables are involved and how[6].

We are going to define for each state $X$ the set of all $defs$ and uses denoted as $adu(X)$ and we will consider each step, from a data flow perspective, as a basic block[9][17].

Predicate uses within a step cause that state to be split in parts: $F$ and $G$ have been split in $F,F_1,F_2$ and $G,G_1,G_2$. This is based on the concept of extended basic block. Note that execution of $F_1$ or $F_2$ within the step $F$ is implicitly determined in the definition of variable $a$ and does not correspond to a surgeon choice during the step $F$. The only choice here would be the execution of $F$ rather than any other eligible step.

These two steps in the example behave differently according to the value of $a$ that, in this example can have two values 1 or 2, anyway both the steps can be considered as extended basic block. In fact the execution flow within the block is deterministic.

We can represent the set of order constraints in a DG as in fig.4(a). Any possible order of execution can be obtained by picking nodes from the DG respecting all dependencies from other step which have not yet been executed. We show two possible CFGs in fig.4(b). For each graph we can see that there are two feasible paths. Both these CFGs respect the set of order constraints explicitly defined by the data flow analysis of each step.

### 3.2 Reducing complexity through design for testability

To reduce the testing effort, development was inspired to general principles of design for testability [18], and in particular to the usage of design patterns that support of effective and efficient verification of functional assumptions through static inspection of code architecture[19][20]. Note that, in so doing, the structure of implementation is conditioned to functional testing objectives.

In particular three major assumptions were supported through adequate and verifiable choices in the implementation structure.
Fig. 3. The adu sets for the example workflow. For each step the set of all definitions and uses of variable is reported.

- The structure of implementation of the workflow model was implemented using the BLUIGS engine. In so doing, the workflow is explicitly encoded into a set of states and a set of dependency rules, whose consistency with the expected specification can be supported by static code inspection.
- Each variable $a \in V$ was restrained to be defined within a single activity of the process:

$$\forall A, B \in S, \forall a \in V, a \in def(A) \land a \in def(B) \rightarrow A = B$$

This constraint was enforced at design level, by making each activity be a class and each global variable be a private attribute of the class were it is defined with public get methods and private set method.
- Consistency in subsequent choices subordinated to the p-use of the same global variable is guaranteed through the verification phase since any inconsistency would result on a test failure caused by missing data or incorrect values.
- The choice of a particular step during the procedure is constraining, i.e. it is not possible to re-execute an already visited operational phase. This assumption is not restrictive in this case, since for instance it has no sense to reacquire a registration point after the cut has been executed.

3.3 Reducing complexity through test equivalence

Under the assumptions enforced at design level, the test suite that guarantees all-uses coverage can be drastically reduced as most test cases turn out to be equivalent.

The dependency DAG defines a partial order between states implicitly defining an equivalence class of CFGs. Based on the example dependency DAG there are 76 possible graphs, or in other
words, 76 ways to complete the process without breaking any order constraint. Considering the
data flow analysis that would make $76 \times 2$ possible du-paths.

Based on this partial specification, any order in which we complete the task without breaking
any constraint can be obtained by picking a node at a time from the dependency DAG following
the rule that we can pick any node that does not depend on other node in the DAG. Explored
nodes are removed from the dependency DAG as showed in fig. 5.

The workflow engine must simply enable the choice of a subset of steps that are eligibles
based on a queue of already explored steps and the set of all dependencies.

Any choice made is thus guaranteed to be a valid workflow since all dependencies are re-
spected. Plus all the possible ways of choice order are equivalent since, at any decision, only a
subset of states is eligible. All nodes in that subset are independent guaranteeing that the order
in which steps are performed is equivalent. A similar concept is the one applied in the case Out-
Of-Order Execution, where data flow information is used in order to optimize the CPU resources
usage, in any case the data flow analysis detect different execution order that do not affect the
final result of computation.

Let be $S_1 \rightarrow S_n$ the sequence of explored states, and be $ES = S_{i_1}, S_{i_2}, \ldots, S_{i_m}$ the set of eligibles states, any sequence $\forall S_{i_k} \in ES$ the sequence $S_1 \rightarrow S_n \rightarrow S_{i_k}$ is equivalent.

Expanding this concept in terms of equivalence between du-paths, let be $S_i$ the step that
contains the definition of a variable $i$, and let be $S_j$ a subsequent step where the value of $i$ is
used. Let be $S_i \rightarrow S_k \rightarrow S_j$ an execution order that execute the du-path between $S_i$ and $S_j$
with respect to variable $i$. For the hypothesis of non-interference, $S_j$ is the only portion of code
where the value of $i$ is modified, meaning that any step $S_x$ executed between $S_i$ and $S_j$, either is
independent from $S_i$, or contains a use of $i$, but for sure will not modify its value. Therefore, a

Fig. 4. In (a) is represented the dependency DAG based on the set of constraints generated by data flow
analysis. In (b) there is an example of two among the many possible control flow graphs that can be
obtained respecting the order constraints. Note that the control flow graphs represent two possible run
time execution of the process in which state F and G have been split considering the run time value of
variables involved in the corresponding $p-use$ in F and G. For each CFG, or in other words for any
legal execution order would require at least two test cases executing the two possible path on the graph.
Fig. 5. An example of execution of three steps (a), (b) and (c) on the dependency DAG. In the first step node C is selected, followed by selection of B and C. Once the third step is complete there are three eligible nodes as next step: A, E and D. Steps F and G are not eligible until their dependencies are verified. Note that dependency DAG defines no relation between nodes in the eligible set.

A test case executed in this path, would produce the same result as in any other possible execution path \( S_i \sim S_y \sim S_j \).

In conclusion, given any of the CFGs that respects all data dependencies, it is possible to build a test suite covering all-du paths, that produce the same coverage of all-du paths in any other order of execution.

### 3.4 Application to our case study

In the application to the case of the Miró system, the equivalence between the possible CFGs allows to build an equivalent test suite based on one on the many possible orders of execution. In this case the results is far more relevant, since it means that the same test suite, and also the corresponding data set for verification, built for a fixed-workflow scenario is still valid. Fig. 6 illustrates the abstraction applied to Miró.

Analyzing the CFG, there are only two variables for which there is at least a p-use in some step. This variables are indicated as 0.1 and 0.2 and consist respectively on: the choice of the prosthesis, and the type of acquisition to use as a reference to evaluate the correct positioning of the femoral prosthesis component. Both the variables can assume 3 values, leading to a minimal test suite built on 3 path for all-uses and 9 paths for all-du paths using the two variable in all the possible combinations. The use of this test suite ensures the execution of all possible behaviors also on SOUP items where unit test level does not provide any form of verification.

Even though, the use of computer assisted surgery is increasing, it still remains a lower percentage compared to traditional technique, and that due also to its lack of flexibility. The
adu(A) = \{ \text{def}(0.1), \text{def}(0.2) \}

adu(B) = \{ \text{def}(1.1), \text{def}(1.2), \text{def}(1.3), \text{def}(D.1), \text{def}(D.2), \text{def}(D.3), \text{def}(D.4),
            \text{c-use}(1.1), \text{c-use}(1.2), \text{c-use}(1.3) \}

adu(C) = \{ \text{def}(2.1), \text{def}(2.2), \text{def}(2.3), \text{def}(2.4), \text{def}(D.5), \text{def}(D.6), \text{def}(D.7), \text{def}(D.8), \text{def}(D.9),
            \text{c-use}(2.1), \text{c-use}(2.2), \text{c-use}(2.3), \text{c-use}(2.4) \}

adu(D) = \{ \text{def}(3.1), \text{def}(3.2) \}

adu(E) = \{ \text{p-use}(0.1), \text{def}(4.1), \text{def}(4.2) \}

adu(F) = \{ \text{def}(5.1), \text{def}(5.2) \}

adu(G) = \{ \text{def}(6.1), \text{def}(6.2) \}

adu(H) = \{ \text{def}(7.1), \text{def}(7.2), \text{def}(7.3), \text{def}(7.4), \text{def}(7.5), \text{def}(7.6), \text{def}(7.7),
            \text{def}(D.14), \text{def}(D.15), \text{def}(D.16), \text{def}(D.17), \text{c-use}(D.4), \text{p-use}(0.1), \text{p-use}(0.2) \}

adu(I) = \{ \text{def}(8.1), \text{def}(8.2), \text{def}(8.3), \text{def}(8.4) \}

adu(J) = \{ \text{p-use}(0.1), \text{def}(D.18), \text{c-use}(D.4) \}

adu(L) = \{ \text{def}(13.1), \text{def}(13.2), \text{def}(D.19), \text{c-use}(D.4), \text{c-use}(D.10) \}

Fig. 6. Miró Integration Test Plan

Total Knee arthroplasty surgical procedure comprise the execution of cuts both to the tibia and the femur, surgeons are used to perform these two phases following one particular order better than the other. A flexible procedure supports surgeons with this choice, making them more confident when passing from the conventional technique to a computer assisted procedure, without the need of modifying their way of conducting the operation.
4 Conclusions

In our case, it has been possible, following this approach, to reuse the integration test suite detected in the previous software version. In this way, any increase in terms of cost for integration test execution has been avoided, adding on the same time the flexibility feature. This advantage is particularly relevant, when considering that integration test phase is the most complex verification phase in this context.

Most of the effort has been put instead, on verifying the aforementioned assumptions about non-interference of procedure steps on the variables, while at the level of unit test a test suite has been introduced to test the new workflow engine, testing the engine on accepting or reject all the possible execution orders.

The last point to verify was the assumption that we can consider a single step as a basic block. This assumption is indeed a big limitation in the case of the registration step, where the set of anatomical points are acquired to build the anatomical reference system. At this level, flexibility allows the surgeon to re-acquire any of the registration points without the need of re-acquiring all values. The use of design patterns oriented for testability can help also to remove the assumption that within the same step a variable can not be re-defined. In other words this would imply the redefinition of a variable \( x \), that can have been already e-used in the same step to compute a variable \( y \), causing the risk of this second variable to be not correctly updated. The information about the dependency between data related to each step, can be used to automatically update all dependent data. This automatic update, has been guaranteed by using an extension of the observer pattern[21], where all variables are encapsulated in a data class, able to notify changes on its internal state and to observe notification from other data object in the same step.

Even though the assumptions, valid in this particular study case, appear to be restrictive, we believe that this kind of approach can be extended to cover more general workflow specifications. Flexibility is a common issue in many kind of workflow applications[11], and this approach can easily be applied to any domain in which a similar workflow modeling is suitable.

In conclusion, this case study reports on how, combining the right integration testing approach, in our case the data flow testing, with some elements oriented to design for testability it is possible to implement major changes on a certified software minimizing the cost of the verification process.

References