Extended Choice Relation Framework for Workflow Testing

Marlon Dumas*
University of Tartu, Estonia
marlon.dumas@ut.ee

Luciano García-Bañuelos
University of Tartu, Estonia
luciano.garcia@ut.ee

Kam-Seng Ho
University of Macau, Macau, China
jackyho.13@gmail.com

Yain-Whar Si
University of Macau, Macau, China
fstasp@umac.mo

Abstract

Testing workflow applications is important given their critical role in an organization's everyday operations. At the same time, systematic workflow application testing requires a significant amount of effort since these applications usually manipulate non-trivial data structures and interact with many other application components. Therefore, devising efficient and effective ways of generating test cases for workflow applications is a highly relevant problem. In this paper, we propose a method for workflow test case generation based on the Choice Relation Framework (CRF). The proposed method aims at reducing the amount of input required from the sofware tester, while allowing test cases to be generated incrementally. To this end, test cases are generated in such a way as to cover the most frequently-occurring workflow execution traces, which are identified based on conditional branching probabilities typically available in workflow models. An empirical evaluation demonstrates that the proposed method significantly reduces the amount of input that testers need to provide in order to produce test cases that cover the most frequent paths of the process.

1 Introduction

A *workflow* (a.k.a. a *business process* in some settings) is a collection of inter-linked activities involving human actors, IT systems and other resources that collectively allow an organization to deliver a product or service or to achieve some other organizational goal. An example of a workflow is the set of activities performed by a company in order to handle a purchase order. This workflow starts when a Purchase Order (PO) is received and ends when the products are shipped and an invoice is produced.

A *workflow application* is an application that assigns work to human actors and other resources and that transfers data across IT systems in order to ensure that a given workflow is performed efficiently. Generally, workflow applications are built starting from a *workflow model* that captures the order in which work (tasks) are performed in a workflow. A typical workflow model is a graph consisting of at least four types of nodes: *tasks*, *events*, *gateways* and *data objects*.[1] Tasks describe units of work that may be performed by humans or IT systems, or a combination thereof. Events denote built-in workflow events such as start and completion of an activity or inputs to be provided by the environment to the workflow application. Gateways capture the flow of execution between tasks and events, therefore establishing which tasks should be enabled or performed after completion of a given task. Finally, data objects denote parameters that are read or updated during the workflow. A simplified example of a workflow model for handling credit card applications is given in Figure 1. This model is captured using the Business Process Model and Notation (BPMN).[2] In this example, all gateways (cf. elements with diamond shapes) are exclusive decision gateways, meaning that they correspond to points in the process where one among multiple outgoing branches is taken. In this model, each gateway is associated with a *data object*. This indicates that the data object is used to make the decision. The outgoing branches

*Work conducted while the author was on leave at University of Macau.

[1] Other types of nodes can also be found in workflow models but we omit them for the sake of simplicity.

[2] http://www.bpmn.org

of each decision are annotated with a percentage value, indicating in what percentage of the time the outgoing branch in question is taken. This information is typically included in high-level workflow models since it is needed to analyze bottlenecks in the workflow. Tasks (represented by rounded-corner rectangles) are also associated with (input) data objects that determine how the task is performed. For example, the amount will determine how the purchase order is checked.

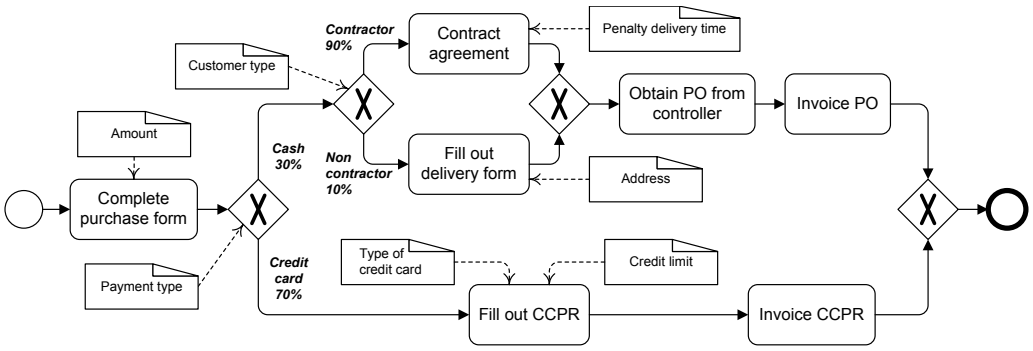

Figure 1: Workflow specification of simplified order management process

Given their central role, the dependability of workflow applications is critical in modern organizations. Accordingly, the verification of such applications is a highly relevant problem in practice. This paper focuses on verifying workflow applications by means of testing.

Test case generation is one of the central tasks in software testing. A large body of research in software testing tackles the problem of automating the generation of test cases based on (high-level) system specifications or source code. The aim is to create methods that minimize the manual effort required from software testers while maximizing coverage with respect to a given criterion.

Partition Testing [9] is a family of techniques for generating test cases from functional specifications. In partition testing, the tester divides the system's input domain into sub-domains according to a property. A key assumption in partition testing is that within each sub domain, the program either produces the correct answer for every element in the sub-domain, or it produces an incorrect answer for every element. Often, multiple sub-domains are inter-linked in such a way that selecting a particular value in one domain, reduces the space of valid elements that can be selected in another sub-domain. For example, in the context of a payment system, it may happen that choosing a value above 2000 for the "amount to be paid" excludes the possibility of selecting "credit card" in the domain "payment instrument" (i.e. payments above 2000 cannot be made by credit card).

The Choice Relation Framework (CRF) [3] is a partition testing technique that takes advantage of dependencies between partitions to reduce the amount of effort required from the software tester. The key ideas of CRF are that sub-domains are defined by means of choices, and that the tester specifies relations between choices. Given the tester-specified dependencies, additional relations are derived automatically. Test cases are generated based on the choices and their relations.

In this paper, we propose an extension of the CRF framework that takes advantage of the information captured in workflow models in order to further reduce the amount of effort required to generate test cases. The proposed extension leverages the control flow logic captured in a workflow model, in order to incrementally generate test cases, in such a way that the most frequent paths in the workflow are given higher priority. The potential benefits of the extended framework are evaluated on a repository of 42 workflow models from industry practice.

This rest of the paper is organized as follows. An overview of CRF and of other related work is given in Section 2. The proposed extension to CRF is then presented and exemplified in Section 3. Section 4 gives a preliminary empirical assessment of the potential benefits of the extended framework. Finally,

Section 5 summarizes the contributions and directions for future work.

2 Background and Related Work

2.1 Choice Relation Framework

Test case generation techniques can be broadly classified into two families: black-box techniques, which rely only on functional specifications, and white-box techniques, which rely on source code analysis. Techniques in-between are called grey-box.

The Category Partition Method (CPM) [9] is a well-known black-box testing technique. The first step in CPM is to decompose the system into relatively independent functional units and to characterize the parameters and environment conditions that affect each functional unit. Based on this analysis, a number of *categories* and *choices* are derived for each functional unit. A category is a parameter or environment condition that affects the execution of the program in a significant way. For example, in the context of a program to sort an array, the size of the array is a property that can be represented as one category. Each category is partitioned into distinct choices. A choice is a set of similar values within a category that are considered as equivalent for the purpose of test case generation. For example, we can partition category *array size* into choices $size = 1$, $size = 2$, $2 < size \leq 100$ and $size > 100$. Choices may be inter-linked, meaning that making a choice in one category affects the possible choices in other categories. Accordingly, the next step in CPM after identifying categories and choices, is to determine relations between choices. Once these relations are determined, CPM provides a technique to generate test frames. A test frame is a set of choices that comply with the constraints captured by the choice relations, and such that each category is represented by no more than one choice in the test frame. Test cases are generated from test frames by selecting one element from each choice. For example, if a test frame contains the choice $2 < size \leq 100$, the system can choose to generate a test case in which $size = 3$. Note that test frames and test choices are quite similar, the difference being that test frames are composed of choices while test cases are composed of concrete input values.

One issue with CPM is that it requires testers to define all possible constraints between choices. The Choice Relation Framework (CRF) [3] alleviates this burden by providing rules that allow certain constraints to be derived from others. This way, the tester only needs to define a subset of the constraints.

For the example given in Figure 1, we can define following categories and corresponding choices:

- "Amount": $0 < a \leq 2000$, $2000 < a \leq 4000$, $4000 < a$.
- "Payment Type": "Cash", "Credit Card".
- "Customer Type": "Contractor", "Non-Contractor".
- "Penalty": 200, 400, 600.
- "Delivery Time": $0 < d \leq 10$, $10 < d \leq 20$, $20 < d \leq 30$.
- "Type of Credit Card": "Gold","Classic".
- "Credit Limit (only applicable to Classic Credit Card)": 2000, 4000.
- "Address": "US", "Non-US".

In CRF, constraints are captured by means of a table where each row and each column corresponds to a choice. A cell in this table represents a relation between two choices. Three types of choice relations are captured in CRF, as exemplified in Table 1:

1. x is fully embedded in y (x $\sqsubset$ y), meaning that choice x implies choice y. For example, choice "Gold" in category "Type of Credit Card" is fully embedded in choice "Credit Card" in category "Payment Type". Thus, if a test frame contains choice "Gold", the choice "Credit Card" is implied in that test frame.
2. x is partially embedded in y (x $\sqsubset_P$y), meaning that choices x and y may or may not co-exist in a test frame. For example, choice "$0 < a$ <2000" in category "Amount" is partially embedded in choice

"Credit Card" in category "Payment Type".

3. x is not embedded in y (x $\not\sqsubset$ y), meaning that any test frame that contains x must not contain y. For example, if the purchase amount is $0 < a < 2000$, the penalty of delivery delay is 200, while if the purchase amount is $2000 < a < 4000$, the penalty of delivery delay is 400 and for $a > 4000$, the penalty of delivery delay is 600. Thus, choice "$0 < a < 2000$" in category "Amount" is not embedded in choices "400" and "600" in category "Penalty".

	Cash	Credit Card	Contractor	non-Contractor	$\$0 < a \leq \$2000$	$\$2000 < a \leq \$4000$	$\$4000 < a$	$0 < d \leq 10$	$10 < d \leq 20$	$20 < d \leq 30$	\$200	\$400	\$600	Gold	Classic	\$2000	\$4000	US	Non-US
Cash	$\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$
Credit Card	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$
Contractor	$\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$
Non-Contractor	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$
$\$0 < a \leq \$2000$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$
$\$2000 < a \leq \$4000$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$
$\$4000 < a$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$
$0 < d \leq 10$	$\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$
$10 < d \leq 20$	$\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$
$20 < d \leq 30$	$\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$
\$200	$\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$
\$400	$\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$
\$600	$\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$
Gold	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$
Classic	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$
\$2000	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$
\$4000	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$
US	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\not\sqsubset$
Non-US	$\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\sqsubset\!\!\!\!\!\!\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\not\sqsubset$	$\sqsubset$

Table 1: Choice relation table of the running example

Chen et al. [3] define a set of rules – e.g. symmetry of non-embedding choices – that are used to derive new constraints given a set of constraints defined by a tester. They show by means of a case study that in practice, this method may reduce the amount of manually-specified constraints by around 40%.

The CRF also enables testers to attach priorities to choices. For example, the tester may decide that choice "Customer Type = Contractor" has a priority of 1, while "Customer Type = Non-Contractor" has a priority of 2 (here a higher number indicates a lower priority). These priorities are used to reduce the number of generated test cases in case there exist resource constraints, which are common in software testing processes. Specifically, the tester can define a Minimally Achievable Priority Level (m) and a Preferred Maximum Number of Test Frames (M). Once the limit M of test frames is reached during the testing process, choices with priority greater than m will not be selected to form further test frames (although further test frames will be generated if certain choices with priority $\leq m$ are not yet included in any test frame). By giving appropriate values to m and M, the tester can control the number of generated test frames in order to fit a resource constraint.

2.2 Workflow Testing

A number of methods for test case generation for workflow applications have been proposed. Yuan et al. [10] propose an extended control flow graph to abstractly represent executable workflow models specified in the Business Process Execution Language (BPEL). Sequential test paths are generated from this abstract representation based on a branch coverage criterion. Next, a constraint solver is used to

detect all infeasible paths and to generate test cases for the feasible paths. In [2], a methodology and tool to generate test cases from BPMN process models are proposed. BPMN models are transformed to Algebraic Petri Nets (APN), which is an extension of Petri Nets with Algebraic Abstract Data Types. Next, all necessary state spaces are produced from the APN. Finally, model checking techniques are applied to build the state space and test cases are generated with their respective oracles. A similar approach is used by Garcia-Fanjul et al. [6] to generate test cases from BPEL specifications. This latter work uses the SPIN model-checking tool to drive the selection of test cases according to a transition coverage criterion. Meanwhile, Mei et al. [8] propose a method for prioritizing test cases in the context of regression testing of BPEL processes. Their method is based on branch coverage criteria that take into account not only the control-flow graph (i.e. which paths are taken) but also the data-flow, which is captured in the XPath expressions in the BPEL process definition and in the WSDL message type definitions attached to such processes.

The above methods are white-box insofar as they take as input an executable specification of the workflow, including data types, branching conditions and data update operations. However, in some scenarios, the available workflow models are high-level models intended for analysis and design purposes, rather than fully executable models. Such high-level models do not contain all the information required by white-box testing methods. Still, high-level workflow models contain valuable information that can be used as input for black or grey-box testing techniques.

Grey-box workflow application testing techniques include that of Zhang et al. [7], which transform a business process model into a dynamic testing model based on extended UML 2.0 activity diagrams. Zhang et al. apply depth-first search to the extended UML 2.0 activity diagram in order to generate test sequences. Test cases are then obtained by applying CPM to the test sequences. Meanwhile, Bakota et al. [1] propose a semi-automatic test case generation method based on CPM. The method consists of four levels: Model level, Path level, Test frame level and Test case level. First, a control-flow graph is derived from the functional specifications. Second, all possible paths are generated from this control-flow graph. Third, test frames are generated based on the paths and the predefined partitions of each category. Fourth, test cases are generated by binding data to their input derived from test frames. The limitation of this method is the lack of an explicit mechanism for capturing relations between choices. As discussed earlier, such relations are critical in order to reduce the amount of input required from the tester. Interestingly, Bakota et al. put forward the idea of prioritizing the generation of test cases in order to focus on the most frequent paths. However, Bakota et al. capture the frequency of each path using a discretized scale by classifying paths into those that are "always" taken, "often" taken, "rarely" taken and "never" taken. In our proposal, we push this idea further by using probability values attached to choice edges of workflow models in order to estimate the probability that an execution of the workflow will take a given path (and thereby its frequency).

3 Extended Choice Relation Framework

The proposed extended CRF exploits the following properties of (high-level) workflow models:

1. Workflow models contain information about choices, data objects (i.e. parameters) involved in these choices and the relations between these choices.
2. Workflow models contain information about the paths (i.e. sequence of tasks) that the workflow application can take.
3. Workflow models, especially those used for quantitative analysis and simulation, contain branching probabilities that can be used to determine the most frequent execution paths of the workflow, and thus of the workflow application.

The first property allows us to prune down the space of choice relations that the tester needs to manually specify. To take advantage of this property, we introduce a distinction between routing and non-routing choices. Routing choices play a role in the execution path taken by workflow application, meaning that they play a role in the decision gateways of the workflow model. For example, in the running example "Payment type = credit card" is a routing choice. In contrast, non-routing choices do not play a role in any decision gateway, but instead they determine how tasks are executed. For example, the "Type of credit card = Gold" will play a role in the performance of task "Fill out CCPR", meaning that this task will be performed differently when this choice holds vs. when it does not hold. In the proposed extended CRF, relations between routing choices are automatically derived from the workflow model as detailed later.

The second property allows us to make the test generation incremental by focusing on one workflow execution path at a time. Since workflow models may include parallel branches (meaning that multiple tasks are executed in parallel or any order) we use the notion of *run* (defined below) to decompose a workflow model into a set of possible executions. A choice relation table is constructed for one run at a time, based on the choices involved in that run only. Therefore, instead of the tester having to determine the entire relation table in a single go before generating test cases, she can focus on the subset of the choice relation table for one run and produce an initial set of test cases for this run before moving on to another run.

Finally, the third property allows us to prioritize the tester's effort by focusing on the most frequent paths (and thus the most critical choices). Specifically, given the branching probabilities in the workflow model, we can compute the probability of execution of each individual run (i.e. the run's frequency). Thus, the tester can start by defining the choice relation table for the run with the highest probability and proceed with other runs incrementally until a given level of coverage is achieved (i.e. until test frames are generated covering at least X % of the executions). In the same vein, we can compute a probability for each choice (whether routing or non-routing). This choice probability can be used by the tester in order to assign a priority to each choice as required by the CRF.

A flowchart of the proposed test case generation method is given in Figure 2. Below we describe each individual step, except for the last step (generating test cases from test frames), which is omitted since it is identical to the last step of the CRF framework.

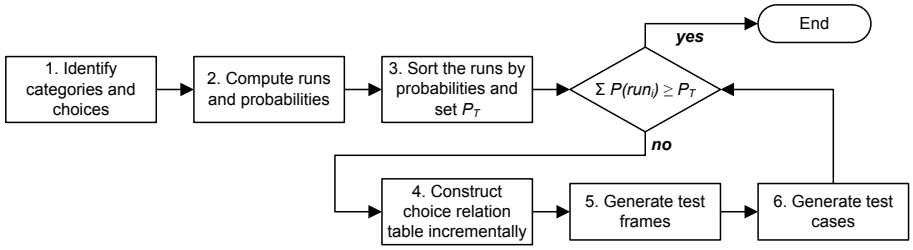

Figure 2: Overall test case generation process

3.1 Step 1: Identify categories and choices

This step is similar to the corresponding step in CPM and CRF. The main difference is that we divide the choices into routing and non-routing choices as defined above (and non-routing categories by extension). Routing choices are extracted from the workflow model. Each XOR-split becomes a category while each edge emanating from the XOR-split in question becomes a routing choice under this category. Next, for each task and event in the model, the tester may designate a number of (non-routing) categories and choices. Note that the start event(s) of the workflow model are also included in the set of nodes for which

non-routing choices are defined. Start events convey the data that is given as input by the environment when a new instance of the workflow model is started.

3.2 Step 2: Compute runs and non-routing choice probability table

A run is a subgraph of an acyclic workflow model comprising the set of all edges traversed in one possible execution of the workflow model. The concept is akin to the notion of *execution path*, but in a run there may be parallel splits and joins (and thus a run may contain multiple paths), whereas a trace is usually defined as a specific path in a graph. To illustrate this concept, we consider the sample process model in Figure 3(a), which contains parallel splits and joins (cf. gateways with "+" symbols). In BPMN, the semantics of such parallel gateways is the following: A parallel gateway with multiple outgoing arcs splits (a.k.a. *AND*-split), initiates multiple parallel branches, while a parallel gateway with multiple incoming arcs (a.k.a. *AND*-join), synchronizes the incoming branches according to a "wait-for-all" semantics. In *XOR*-split, among a number of branches originated from that split, only one branch which satisfies the pre-defined condition will be chosen. A run of this process model is a connected subgraph of the process model that contains the start (entry) and the end (exit) node, and such that at most one outgoing edge of each *XOR*-split is represented. For example, Figure 3(b) is one of the runs of Figure 3(a). Two other runs can be constructed: one by taking choice edges p_1 and p_4 and another by taking choice edge p_2, where a choice edge is defined as an edge whose source node is an *XOR*-split.

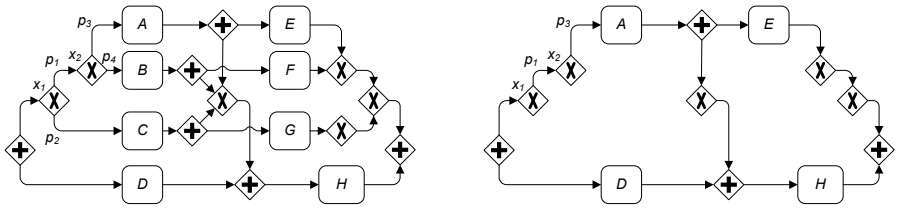

Figure 3: Sample workflow model (a), and one possible run (b)

Algorithm 1 computes the set of runs of an acyclic process model and their execution probabilities (the cyclic case is discussed below). The algorithm relies on a depth first search (DFS) traversal of the input model. Firstly, the *entry* node is put in stack *S* (line 3). As per the classical DFS, the algorithm proceeds by iterating as long as *S* is not empty (lines 4-31). In each iteration, the algorithm retrieves the top element of stack *S* (line 5), namely *curr*, and pushes every successor *succ* of *curr* into *S* (lines 15-17). When a node is visited, it is copied to V_{run} (line 6). Similarly, edges are copied to E_{run} whenever they are traversed (line 16). V_{run} and E_{run} define the "currently-traversed run" of the workflow model. Join nodes have multiple predecessors and hence deserve special attention. A parallel (*AND*) join requires all its predecessors to be visited before it is processed. In contrast, a merging (*XOR*) join requires only one predecessor to be previously visited in order to be processed. Those conditions are checked in line 7. Also choice (*XOR*) splits require a special treatment. Whenever a choice (*XOR*) split is found (line 8), it is inserted into a separate stack *XORSplitStack* (line 9) and one of its successors is selected and marked (lines 10-11). Then the traversal continues accordingly (lines 12-13). In this way, only one choice edge per *XOR*-split is included in the currently-traversed run. When the exit node is reached, which corresponds to the "else" statement in line 18, the algorithm adds the current run to set *Runs* (line 19). The probability associated with the run is calculated by multiplying the probabilities associated with every choice edge in the run. In order to compute the next run, the algorithm makes a "backtrack" by selecting the split at the top of *XORSplitStack* (line 21). If this split has at least one successor not

"marked", the algorithm removes all nodes and edges in the current run that come after such split node (line 23), selects one unvisited successor (line 24), and proceeds with the DFS traversal (lines 26-28) starting from that node. If all the successors of the split at the top of *XORSplitStack* have been visited, the algorithm removes the top split (lines 30-31) and repeats the analysis with the remaining nodes in the stack, if any. When *XORSplitStack* is empty, it means all choice edges have been considered and the algorithm has generated all runs.

Algorithm 1: Compute runs

Input: $G = (V, E)$ – Input process graph (must be acyclic)
$entry, exit$ – Entry and exit nodes of the graph, respectively
Output: $Runs : 2^{(V,E,P)}$ – Set of runs associated with the input process graph

```
1  S, XORSplitStack, V_run, E_run, Runs ← ∅
2  foreach v ∈ V : typeOf(v) = XORSplit do marked[v] ← ∅
3  push(S, entry)
4  while S ≠ ∅ do
5      curr ← pop(S)
6      V_run ← V_run ∪ {curr}
7      if predecessorsOf(curr) ⊆ V_run ∨ typeOf(curr) = XORJoin then
8          if typeOf(curr) = XORSplit then
9              push(XORSplitStack, curr)
10             succ ← select any successorsOf(curr)
11             marked[curr] ← {succ}
12             E_run ← E_run ∪ {(curr, succ)}
13             push(S, succ)
14         else if curr ≠ exit then
15             foreach (curr, succ) ∈ E do
16                 E_run ← E_run ∪ {(curr, succ)}
17                 push(S, succ)
18         else
19             Runs ← Runs ∪ {(V_run, E_run, Π_{e∈E_run} Pr(e))}
20             while XORSplitStack ≠ ∅ do
21                 top ← peek(XORSplitStack)
22                 if successorsOf(top) ≠ marked[top] then
23                     removeSuffixPath(top, V_run, E_run)
24                     succ ← select any successorsOf(curr) : succ ∉ marked[top]
25                     marked[top] ← marked[top] ∪ {succ}
26                     E_run ← E_run ∪ {(top, succ)}
27                     push(S, succ)
28                     break
29                 else
30                     marked[top] ← ∅
31                     pop(XORSplitStack)
32     return Runs
```

The worst-case complexity of Algorithm 1 is exponential of the input model, since potentially one run needs to be generated for each combination of choice edges stemming from different XOR-splits (thus, 2^n combinations in the worst case where n is the number of XOR-splits). This complexity is inherent given that the number of runs of a workflow model is exponential in the worst case. However, the empirical evaluation reported below shows that this worst-case complexity is not problematic in practice.

If a workflow model contains loops, it may have an infinite number of runs since a loop may be taken any number of times. This is a general problem in the field of software testing, which is typically addressed by unfolding each loop according to a 0-1 criterion, meaning that each node in the loop is copied either zero times or one time in order to expand the loop into an acyclic control-flow structure that represents a subset of the possible executions of the original loop (cf. [10]).[3]

3.3 Step 3: Sort runs and set threshold

In this step runs are sorted in descendingly according to their probability. The user can set a coverage threshold $P_T \in [0..1]$. Steps 4, 5 and 6 of the method are performed for one run after another until the sum of the probabilities of the runs covered by the testing process is at least equal to P_T. In other words, P_T indicates the required minimum level of coverage.

3.4 Step 4: Incrementally construct choice relation and choice priority tables

Given a run, the purpose of this step is to construct the fragment of the choice relation table corresponding to the run. To this end, we identify all categories/choices that are involved in at least one task appearing in the run. The tester is then asked to define the choice relations between non-routing choices and between routing choices and non-routing ones, using the method defined in CRF [3]. Relations between routing choices do not need to be defined, since all routing choices associated with the choice edges of the run must be included in every test frame generated for the run. Not including one of the routing choices in a test frame would mean that the run is not executed from start to end, but we are interested in testing end-to-end executions of the workflow model.

In the running Purchase Order example, the run shown in Figure 4 has the highest probability (0.7). This run has one routing choice ("PaymentType = Credit Card") and seven non-routing choices ("$0 < a \leq 2000$", "$2000 < a \leq 4000$", "$4000 \leq a$", "CardType = Gold", "CardType = Classic", "CreditLimit = 2000" and "CreditLimit = 4000". Table 2 shows the choice relation table after defining the choice relations for the run. If we look at the cells below the diagonal of this table, we note that only 23 choices (out of a total of 171 cells below the diagonal of the table) need to be determined for this run. The cells above the diagonal of the table can be derived using the rules in [3] while the cells in the diagonal always contain fully-embedded relations. This example illustrates that the proposed extended CRF allows testers to start producing test cases after defining a small fraction of possible choice relations.

As the tester moves from one run to another, choice relations defined for previous runs are reused, while additional choice relations are defined as required by the new runs. For example, the relations between the choices corresponding to category Amount (i.e. $0 < a \leq 2000$, "$2000 < a \leq 4000$", "$4000 < a$") and choice "PaymentType = Credit Card" are reused when constructing the choice relation table for subsequent runs.

Choice priorities are also automatically extracted from the analysis of the workflow model. Specifically, given that we know the probability (i.e. frequency) of each run, we compute the frequency of each task T by summing up the frequency of each run in which T appears. Give the frequency $f(T)$ of a task

[3] Our current implementation uses an unfolding technique that works for workflow models containing single-entry, single-exit loops only. However, extensions can be envisaged for workflow models with multi-entry and multi-exit loops by using loop re-writing techniques.

T, the priority of each choice attached to task T is $\lceil P \cdot (1 - f(T)) \rceil$ where P is the tester's chosen highest ordinal priority value (which corresponds to the lowest priority since lower priority numbers indicate higher priority). Using a similar approach we can attach priorities to non-routing choices. Naturally, the tester can override the choice priorities computed in this way, in order to take into account (for example) the importance of a choice from the perspective of the potential cost of a defect in a particular task of the workflow model.

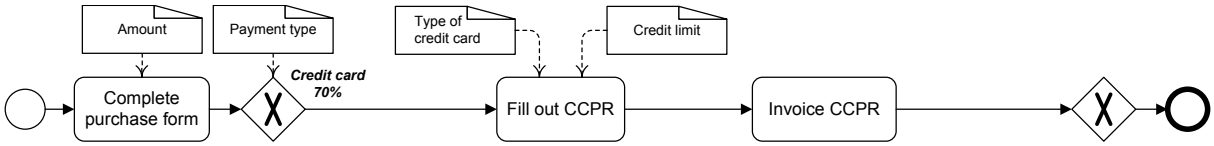

Figure 4: Run of the Purchase Order workflow model with highest probability.

		rc				nrc															
		Cash	Credit Card	Contractor	non-Contractor	$\$0 < a \leq \$2000$	$\$2000 < a \leq \$4000$	$\$4000 < a$	$0 < d \leq 10$	$10 < d \leq 20$	$20 < d \leq 30$	\$200	\$400	\$600	Gold	Classic	\$2000	\$4000	US	Non-US
rc	Cash	No need to define choice relation																			
	Credit Card					$\sqsubseteq_p$	$\sqsubseteq_p$	$\sqsubseteq_p$							$\sqsubseteq_p$	$\sqsubseteq_p$	$\sqsubseteq_p$	$\sqsubseteq_p$		
	Contractor																				
	Non-Contractor																				
nrc	$\$0 < a \leq \$2000$		$\sqsubseteq_p$			$\sqsubseteq$	$\not\sqsubseteq$	$\not\sqsubseteq$							$\sqsubseteq_p$	$\sqsubseteq_p$	$\sqsubseteq_p$	$\sqsubseteq_p$		
	$\$2000 < a \leq \$4000$		$\sqsubseteq_p$			$\not\sqsubseteq$	$\sqsubseteq$	$\not\sqsubseteq$							$\sqsubseteq_p$	$\sqsubseteq_p$	$\not\sqsubseteq$	$\sqsubseteq_p$		
	$\$4000 < a$		$\sqsubseteq_p$			$\not\sqsubseteq$	$\not\sqsubseteq$	$\sqsubseteq$							$\sqsubseteq_p$	$\not\sqsubseteq$	$\not\sqsubseteq$	$\not\sqsubseteq$		
	$0 < d \leq 10$																				
	$10 < d \leq 20$																				
	$20 < d \leq 30$																				
	\$200																				
	\$400																				
	\$600																				
	Gold		$\sqsubseteq$			$\sqsubseteq_p$	$\sqsubseteq_p$	$\sqsubseteq_p$							$\sqsubseteq$	$\not\sqsubseteq$	$\not\sqsubseteq$	$\not\sqsubseteq$		
	Classic		$\sqsubseteq$			$\sqsubseteq_p$	$\sqsubseteq_p$	$\not\sqsubseteq$							$\not\sqsubseteq$	$\sqsubseteq$	$\sqsubseteq_p$	$\sqsubseteq_p$		
	\$2000		$\sqsubseteq$			$\sqsubseteq_p$	$\not\sqsubseteq$	$\not\sqsubseteq$							$\not\sqsubseteq$	$\sqsubseteq$	$\sqsubseteq$	$\not\sqsubseteq$		
	\$4000		$\sqsubseteq$			$\sqsubseteq_p$	$\sqsubseteq_p$	$\not\sqsubseteq$							$\not\sqsubseteq$	$\sqsubseteq$	$\not\sqsubseteq$	$\sqsubseteq$		
	US																				
	Non-US																				

Table 2: Choice Relation Table of Purchase Order for Run 1

3.5 Step 5: Construct test frames

Once the choice relation table corresponding to a run has been defined, the test frames for this run can be generated using the method outlined in [3]. We recall that a test frame is a set of choices that comply with the constraints captured by the choice relations, and such that each category is represented by no more than one choice. In the proposed extended CRF, all routing choices in the run are included in every test frame, for the reasons discussed in Step 4.

4 Evaluation

In order to evaluate the scalability and potential usefulness of the proposed extended CRF, we conducted an empirical study using a collection of models extracted from the IBM BIT process library [4]. This is a library of real-world process models from various sources publicly available for research purposes [5]. The library is organized into three collections labeled A, $B1$, $B2$, $B3$ and C. For this evaluation, we used the models in collection A because all models in this collection are annotated with choice probabilities. We further trimmed the collection in order to eliminate models that were semantically incorrect (e.g. models with deadlocks) and 3 models containing multi-exit loops, which our current prototype implementation cannot unfold. In this way, we ended with a dataset of 42 models.

We first tested the scalability of the proposed technique for extracting runs and constructing their probabilities and associated categories/choices. In this test, we recorded the overall execution time, excluding the time to parse the input XML file. We observed an average execution time was in the order of milliseconds, with the maximum execution time being less than 6 milliseconds.[5] This provides some evidence that the exponential worst-case complexity of computing all runs of a model does not manifest itself in practice. In fact, the models had 6.8 runs on average with a maximum of 36 runs.

Next, we conducted a test to assess the potential usefulness of the extended CRF, specifically its ability to prune the set of choices the tester needs to analyze to achieve a certain level of coverage – where the coverage of a set of test case refers to percentage of actual execution traces of the workflow that are covered by the test cases. To this end, we computed for each level of coverage (in steps of 10%) the average percentage of tasks for which the tester needs to define non-routing choice relations in order to achieve this level of coverage. The results are plotted in Figure 5. The figure shows that, above a 50% coverage level, the effort required from the tester is close to the amount of tasks the tester needs to analyze. For example, 90% coverage is achieved by defining choice relations for 80% of tasks.

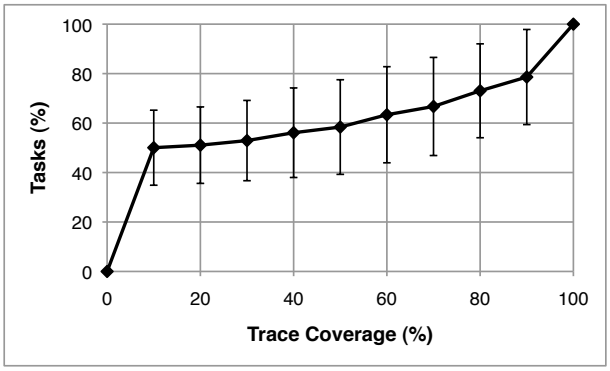

Figure 5: Average percentage of tasks required to achieve a given trace coverage

5 Conclusion

This paper introduced an extension of the Choice Relation Framework [3] for workflow applications that makes use of information contained in workflow models in order to prune the space of choice relations that the tester is required to define, and to make the definition of the choice relation table incremental and

[4] Available at: `http://www.zurich.ibm.com/csc/bit/downloads.html`

[5] This test was performed on a laptop with a dual core Intel processor, 2.53GHz, 4GB memory, running Microsoft Vista and Oracle Java Virtual Machine version 1.6 (with 512 MB of allocated memory).

prioritized – starting with the most frequently-occurring choices. An empirical evaluation demonstrated the potential usefulness of the approach.

As many test automation methods based on control-flow graphs, the proposed method relies on an unfolding of cycles, such that test cases are generated for the execution paths where the body of each loop is repeated zero or one time. In our current prototype, we implemented an unfolding method for single-entry, single-exit loops, which limits the method's applicability since process models in practice sometimes contain unstructured loops. A possible direction for future work is to develop unfolding techniques for process models. Process models may contain parallel branches, which makes the problem of unfolding loops for process models different to that of unfolding loops in classical flowcharts [4].

Another direction for future work is to conduct case studies or controlled experiments to assess the usability of the proposed method in practice. From a usability perspective, a particular point that deserves further attention is how to best guide testers in the identification of categories and choices.

Acknowledgments. This work is funded by the ERDF via the Estonian Centre of Excellence in Computer Science, the Estonian Science Foundation, and University of Macau's Visiting Scholars Program.

References

[1] Tibor Bakota, Árpád Beszédes, Tamás Gergely, Milán Imre Gyalai, Tibor Gyimóthy, and Dániel Füleki. Semi-automatic test case generation from business process models. In *11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering (SPLST'09 & NW-MODE'09)*, pages 5–18, 2009.

[2] Didier Buchs, Levi Lucio, and Ang Chen. Model checking techniques for test generation from business process models. In Fabrice Kordon and Tullio Vardanega, editors, *14th Ada-Europe International Conference on Reliable Software Technologies*, volume 5570 of *Lecture Notes in Computer Science*, pages 59–74. Springer-Verlag Berlin Heidelberg, 2009.

[3] Tsong Yueh Chen, Pak-Lok Poon, and T. H. Tse. A choice relation framework for supporting category-partition test case generation. *IEEE Trans. Software Eng.*, 29(7):577–593, 2003.

[4] Marlon Dumas, Artem Poylyvyanyy, and Luciano García Bañuelos. Unraveling unstructured process models. In *Proceedings of the 2nd Int. Workshop on the Business Process Model and Notation (BPMN)*, Potsdam, Germany, October 2011. Springer.

[5] Dirk Fahland, Cédric Favre, Jana Koehler, Niels Lohmann, Hagen Völzer, and Karsten Wolf. Analysis on demand: Instantaneous soundness checking of industrial business process models. *Data Knowl. Eng.*, 70(5):448–466, 2011.

[6] José García-Fanjul, Javier Tuya, and Claudio de la Riva. Generating test cases specifications for BPEL compositions of web services using SPIN. In *International Workshop on Web Services-Modeling and Testing*, pages 83–94, 2006.

[7] Zhang Guangquan, Rong Mei, and Zhang Jun. A business process of web services testing method based on UML 2.0 activity diagram. In *Workshop on Intelligent Information Technology Application*, pages 59–65. IEEE, 2007.

[8] Lijun Mei, Zhenyu Zhang, W. K. Chan, and T. H. Tse. Test case prioritization for regression testing of service-oriented business applications. In *Proceedings of the 18th International Conference on World Wide Web (WWW), Madrid, Spain*, pages 901–910. ACM, April 2009.

[9] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[10] Jun Yan, Zhongjie Li, Yuan Yuan, Wei Sun, and Jian Zhang. BPEL4WS unit testing: Test case generation using a concurrent path analysis approach. In *17th International Symposium on Software Reliability Engineering*, pages 75–84. IEEE, 2006.