Contract-based test generation for data flow of business processes using constraint programming

A.Jimenez-Ramirez, R.M. Gasca and A.J. Varela-Vaca
Computer Languages and Systems Department
Quivir Research Group
ETS Ingeniería Informática, Avd. Reina Mercedes s/n, 41012
University of Seville, Seville, Spain
{ajramirez , gasca, ajvarela}@us.es

Abstract—**The verification of the properties of a business process (BP) has become a significant research topic in recent years. In the early stages of development, the BP model (e.g. BPMN, EPC), the BP contract (task contract, regulations and laws, business rules), and the test objectives (requirements) are the only elements available. In order to support the modellers, automatic tools must be provided in order to check whether their business processes are in line with the BP contract.**

This paper proposes a new business process called the automatic test-case generator to automate the generation of test cases and verify that a BP has the intended functionality (semantic conformance). This generator is analysed, designed and implemented by taking into account the following tasks: Annotation of the BP model with the business process contract, calculation of the various data flow paths, transformation of these data flow paths into SSA form, and a modelling of a constraint satisfaction problem (constraint programming) of the BP contract for all data flow paths. The execution of this business process generates the test cases automatically.

Keywords:Test Case; BPMN; constraint programming; SSA Form; UML Testing Profile.

I. INTRODUCTION

The verification of business process properties has become an active research topic in recent years. The model of a business process (BP) is determined through the attempt by the modeller to satisfy the BP specification. This model must also comply with established regulations and laws, business rules and standards. Our proposal is the automatic generation of test cases to support these modellers in the early stages of the BP development by checking whether their models are correct according to the predefined specifications (BP contract). These test cases would ease the verification of the soundness, completeness, and performance of the BP models.

Various techniques have been proposed for the verification of syntactic and semantic properties of BPs and of the functionality of the contained activities. A formal model of the BP could be a first option, however, since this is very difficult to obtain, one alternative procedure could be the testing of the business process. The major disadvantage is that, through testing, only the presence of errors can be detected, and not their absence. A test suite must have a significant number of test cases which should be automatically generated due to the difficulty in creating these tests manually. Automation of this task also saves time and expense during development. In most studies to date, these tests are generated in the final phase of the BP development at which point the errors are either very difficult or impossible to correct. Earlier studies propose the creation of a model of the BPEL process for the generation of test cases from this model. In order to automate this procedure, various alternatives are considered, such as concurrent path analysis [4], graph search [5], model checking [2][3], mutation analysis [6] and High Petri Nets [1]. As mentioned, these approaches focus on the test-case generation problem, and the majority use an execution language, such as BPEL [1][2][4][5][6][18]. Only a few approaches are focused on a more abstract notation, such as BPMN and EPC: Bakota *et al.*[19] proposes a semi-automatic process to generate test cases for models expressed in EPC by using the category-partition method (CPM). However, this process requires intervention from the Test designer to extend the EPC model with test information.

This work is focused on a new approach to the problem of verifying the soundness of BPs. We propose a systematic approach to the generation of test cases and to the identification of the conditions necessary to verify a successful and acceptable implementation of the BP contract. The BP model (e.g. BPMN, EPC), the BP contract, and the test objectives (requirements) are the

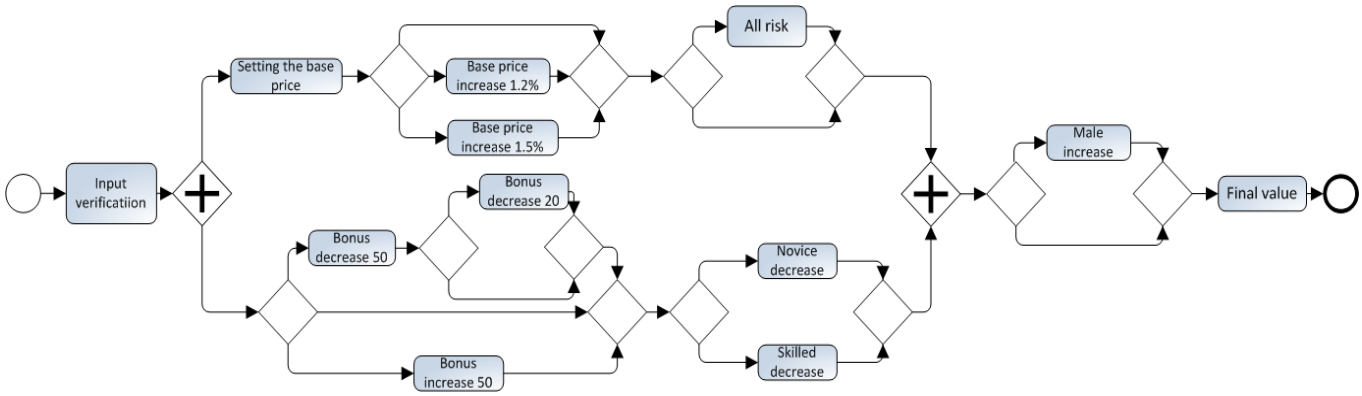

Figure 1. BPMN Model of example

only elements available. In this scenario, it is impossible to access the code of the services of the BP. For the automation of the test-case generation, a new business process is analysed, designed, and implemented where: the BP model is annotated with the contract, the different data flow paths are calculated, the SSA form transformation is carried out over the data flow variables, and the problem of the test generation is modelled as a constraint satisfaction problem (CSP). The constraint programming paradigm permits the efficient and automatic execution of these models for the generation of the corresponding test cases. Similar CSP models have also been considered in previous work: by modelling the functional correctness of other systems using preconditions and post-conditions [7]; and also by significantly improving the diagnosability of the system [8].When the test cases are generated for a business process, their corresponding test procedures must be designed by using test scripts which are then saved in a database. It is supposed that the execution of these test scripts is supported by a business process management system, and the tasks in the business process are application centric (i.e., performed automatically by an application, some software agent, etc). The business rules, the BP model, and contract are all assumed to be correct.

The paper is organized as follows: Section II introduces an example as an incentive for the generation of a suite of test cases for business processes; Section III presents the formal definitions and notation for the automatic generation of test cases; in Section IV, a practical and real business process is developed; Section V shows experimental results that are discussed together with the theoretical results; and, in the final section, conclusions are drawn and future work proposed.

II. MOTIVATING EXAMPLE

In order to generate the test cases for a BP in a BPMS, the model of the business process (BPMN, EPC,…) and the specification of the different tasks that need to be tested must first be outlined. In Figure 1, the BPMN model of a business process for the calculation of the cost of motorcycle insurance is described.

The process begins with the “Input verification” task. This task verifies that the inputs of the process are correct. That is, the “driver’s age”, “horsepower”, “gender”, “kind of insurance” and “age of driver’s licence” have normal values. In addition, this task initializes the “basePrice” and “bonus” (a percentage) variables.

The process has two main parallel branches. One branch calculates the value of the “base price” variable. This variable depends on the inputs “horsepower” and “kind of insurance”. The other branch calculates the value of the variable “bonus” which depends on the value of “driver’s age” and “age of driver’s licence”.

The two variables are then modified by the value of the input ‘gender’. The final value is “insurancePrice”, which is calculated by the following expression: “insurancePrice=basePrice*bonus”.

Each task has an associated behaviour with respect to the data managed (*postconditions*) and also has certain associated needs (*preconditions*). BPMN cannot express these variables, and are hence presented in Table I.

As stated above, the inputs are: “age”, “horsepower”, “gender”, “kind of insurance”, and “age of driver’s licence”. If a user enters the value ‘-1’ as “age”, then the process would end with a default failure value or other behaviour that expresses a fault. As described in the table, the task “Input verification” executes the verification “age>0”, and hence generates an error output if this need remains unsatisfied, which in turn leads to the behaviour explained above.

Furthermore, business rules can also exist within the organization. In this example, only one business rule is considered and is described as follows: “$200 \leq$ insurancePrice ≤ 1800”. This business rule is intended to limit the upper and lower values of the company insurance premiums and is applied to the business processes of the company as a whole.

This specification is the BP contract.

TABLE I. CONTRACT OF THE EXAMPLE

Task name	Task specification	Task requirements
Input verification	basePrice=0; bonus=0;	age >0; horsepower>0; age of driver's licence>0; gender Є ["male","female"]; kind of insurance Є ["all risk","third party"];
Setting the base price	basePrice=300;	none
Base price increase 1.2%	basePrice*=1.2;	horsepower>48
Base price increase 1.5%	basePrice*=1.5;	horsepower >98
All Risk	basePrice+=650;	kind of insurance="all risk"
Bonus decrease 50	bonus -=0.50;	none
Bonus decrease 20	bonus -=0.20;	none
Bonus increase 50	bonus+=0.50;	none
Novice decrease 20	bonus -=0.20;	age of driver's licence<2
Skilled increase 50	bonus+=0.50;	age of driver's licence≥2
Male increase	basePrice+=50; bonus -=0.10;	gender="male"
Final value	insurancePrice =basePrice*(1-bonus);	bonus ≥-100; bonus ≤100;

As can be observed above in Figure 1, the model suffers from a lack of expressivity in terms of the functionality of the tasks, and for this reason our approach enhances the expressivity in a BP.

III. FORMAL DEFINITIONS

By taking the OMG standard UML Testing Profile [9] into account, the testing artefacts can be documented. Our approach works with certain definitions introduced in the OMG document, in addition to the following definitions:

A. BP Model (BPM)

This is a graphical model of the business process represented in the BPMN standard. Any elements in the *BPM* are identified by an attribute *idElement*. Commonly *idElement* is the task name or inner identity of a link.

B. Business Rules (BR)

These are invariants over the variables of the data flow of the *BPM*, and are represented as a set of logical constraints or rules over variables of the data flow. For all the instances of the *BPM*, these rules must be satisfied during execution. Business rules are shared throughout the whole organization and hence inapplicable rules may exist. These inapplicable rules are discarded in the algorithm, for example, given a set of business rules $\Theta = \{R_1, R_2, \ldots, R_n\}$, only subset ϑ is related to a given business process P_1, $\forall R_i \in \vartheta$, which satisfies the requirement that $Var(R_i) \subseteq Var(P_1)$, where R_i is a logical relation between variables of the data flow of the process P_1, and $Var(X)$ represents the set of variables in X.

C. Business Process Contract (BPC)

In this work, *contract* is taken to mean a set of constraints applied over the data flow in certain parts of the *BPM*. Each constraint *C* is a logical relation between variables of the data flow of the *BPM*. These constraints are associated to a task or link, and three types of these constraints are defined here: task preconditions, task postconditions, and gateway activation conditions. Preconditions have to be evaluated before the associated task is executed; postconditions have to be evaluated after; and gateway activation conditions are evaluated to determine which link is taken after a gateway. *BPC* also defines the input parameters of the *BPM* and the output parameters and sets the type of each parameter. We represent *BPC* as a tuple $<I, O, \rho, P, \Gamma>$ where *I* and *O* are sets of tuples <*Parameter, Type*> indicating the input and output parameters respectively; ρ, *P* and Γ are sets of tuples <*idElement, C*> representing the logical relation between variables (*C*) associated to an element in the *BPM* identified by *idElement*. These elements indicate preconditions, postconditions and activation conditions of gateways, respectively.

D. Objective (Obj)

Objectives are defined for the classification of the test cases. Each test is normally developed in order to achieve various objectives. This work describes the "*Complete Cover*" objective, that is, every single test case generated by the process is aimed at achieving this one objective. The "*Complete Cover*" objective is intended to execute all the tasks in the process. Although "*Complete Cover*" is the main objective, our approach is designed to be extended, and hence any objective could be defined through its implementation and each objective can be chosen by a parameter.

E. Business Process Under Test (BPUT)

The System Under Test (SUT) element considered in the UML Testing Profile document is a set of black-box components that are to be tested. This work considers the *BPM* as a unique black-box component, and hence the SUT is the complete *BPM,* and we rename it the Business Process Under Test, *BPUT*. The public interface of the *BPUT* consists of one operation where the inputs and outputs are specified in Section II. In the example of Figure 1, the *BPUT* operation (UML modelled operation) is "*calculate_insurance"* and has five input parameters which are the external inputs of any task in the process. The UML diagram is shown in Figure 2. This approach assumes that the inputs are available for its corresponding task when it is necessary. As shown in Figure 2, this *BPUT* operation has only one output, which is an integer value, and hence *BPUT* can be represented as a tuple <*BPname, INs, OUTs*> where *OPname* is the business process name, and *INs* and *OUTs* are the sets of inputs and outputs respectively.

F. Path

A path represents a sequence of tasks in the *BPM* corresponding to an instance of its execution. Each path begins with a start task, and finishes with an end task. Parallel paths can be contained in a path since parallel gateways are considered in this work. Therefore, in general, a path consists of a list of *idElements* or parallel subpaths. A subpath has the same representation as a path.

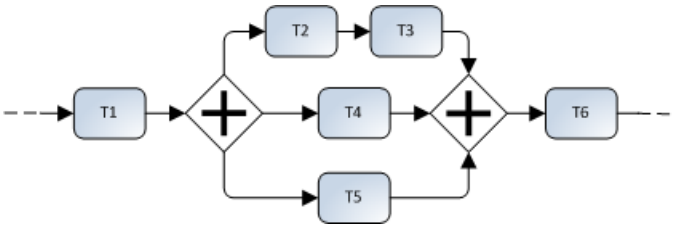

Figure 3. Parallel path example

Figure 3 shows three parallel paths that can be executed at the same time due to their position between parallel gateway elements. The proposed representation in this approach is <…, T1, {<T2,T3>, <T4>, <T5>}, T6,…>.

G. Test Case Data

This element contains the necessary information to generate a test case. Test case data is represented by the tuple <*id,Objs,Ins,Outs,FLog*>. Each tuple defines a test case identification (*id*), a set of input values (*Ins*) and a set of expected output values (*Outs*) for the declared parameters in the *BPC,* i.e. $Ins=\{<in_1, valIn_1>, <in_2, valIn_2>, ..., <in_n, vaIn_n>\}$ and $Outs=\{<out_1, valOut_1>, <out_2, valOut_2>...<out_n,valOut_n>\}$, where in_i is the name of the input parameter *i*, $valIn_i$ is the value associated to this parameter, out_i is the name of the output parameter *i,* and $valOut_i$ is the value associated to this parameter. Furthermore, each test case follows a set of objectives (*Objs*). As explained earlier, our approach is an attempt to achieve this objective in an automatic way. One test case for the example obtains a value for these five inputs, and indicates the expected value for the output. Each tuple of test case data also contains the path in the file system where the execution results of the test case are to be stored (*FLog*).

H. Test cases

As defined in the UML Testing Profile, there are certain behaviours that indicate how different test components stimulate the *BPUT*. In this work, these test components are implemented as a user-like service that executes the specified UML operation of the *BPUT*. When a generated test case is executed, a verdict will be returned indicating "*pass"* or "*fail"* (according to the UML Testing Profile).

As shown in Figure 5, the normal behaviour of a test case evaluates a "*pass*" action. For any non-covered case, the "*fail*" action is triggered and the test finishes.

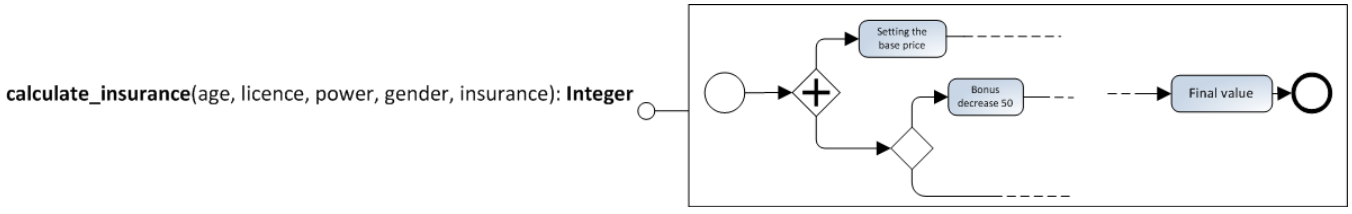

Figure 2. BPUT specification

Each test case follows the same behaviour shown in Figure 5, only input and output values are modified as indicated by each instance of test case data. Therefore our approach has to maintain only one structure in order to execute all test cases generated for this *BPUT* operation.

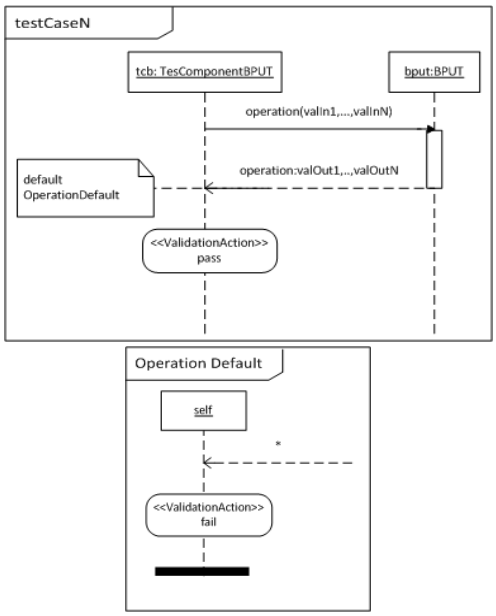

Figure 5. UML specification for Test-Case Behaviour

IV. TEST-CASE DATA GENERATION PROCESS

This work assumes that the process is syntactically error-free. Certain approaches exist which are focused on that objective [10]. A validation task could be performed as the first task of the Test-Case Data Generation Process (*TCDGP*), but remains without discussion in this paper.

As shown in Figure 4, our approach divides the *TCDGP* into four tasks:

- Process Annotation. In this task the *BPM* is annotated. These annotations are provided by the *BR* and *BPC* elements. The process annotations are the graphical representation of *BR* and *BPC* with the *BPM*. This approach does not require the complete process to be annotated. The more annotations the process has, the more useful the test cases will be.

  The contract is formed by the annotations. These annotations give information about idElement, precondition, postcondition, and/or activation conditions, expressed as logical constraints. This paper considers that tasks are service-like elements which have only one operation. Hence, annotations describe the actual operation .

- Path Calculation. This task obtains the various paths in the process. Each instance of generated test case data must correspond to one path.

- SSA Form Transformation. This task makes the transformation into a constraint satisfaction problem (CSP) using SSA.

- Test Case Data Generation. The CSP corresponding to each path has to be solved. This task generates the inputs and expected outputs which are used to define the test case. Our approach uses constraint programming due to its efficiency in combinatorial problems [11].

  This task generates a list of test case data *(<Obj,In,Out,FLog>)* corresponding to an objective (*Obj*), i.e. the difference in the data generated depends on the objective provided for the process.

A. Process Annotation

BPMN provides a certain type of information (or artefact), called annotation, that can contain text. This approach uses this feature to bring more semantics to the processes. These semantics are described through

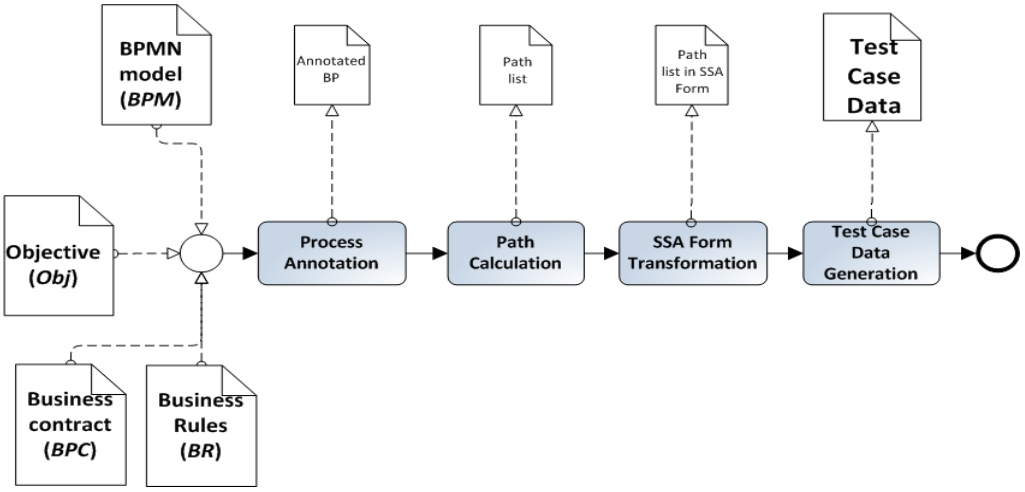

Figure 4. Automatic Test-Case Data Generator Process

constraints. The constraints are specified in a formal language such as the JML standard [12] which specifies contracts for Java programs.

The annotations of the contract must be differentiated from the other annotations, and hence the inner text must contain an identifier. Our approach uses the word *"<contract>"* as the identifier. Three kinds of annotations are distinguished: task annotations, gateway annotations, and start/end annotations. In all cases, these annotations contain constraints over the variables of the data flow. Furthermore, this approach considers another kind of element that contains the business rules (*BR*) of the organization.

All the necessary information for this task is extracted from the *BPC* and *BR* elements provided for the process. Once the *BPM* is annotated, *BPM*, *BPC* and *BR* are no longer required. *Annotated BP* is a rich user-friendly representation, and hence it can be dealt with by any business expert.

1) Start/end tasks

The annotations of the first and last nodes are the most important. They contain information solely about the process inputs and outputs. No constraints are contained in these annotations.

Automatic inference of the types of variables can be performed, but remains outside the scope of this work.

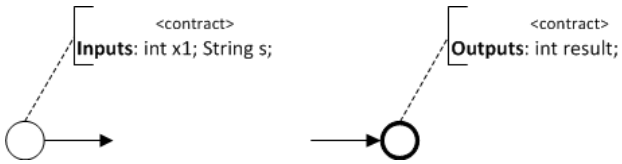

Figure 6. Start/end task annotations

Our approach extracts these annotations from the *I* and *O* elements of the *BPC<I, O, ρ, P, Γ>*. The *Annotated BP* representation for start/end tasks is shown in Figure 6.

2) Tasks

Task annotations describe the preconditions and postconditions. These elements are specified as constraints over variables of the data flow. The graphical representation is shown in Figure 7.

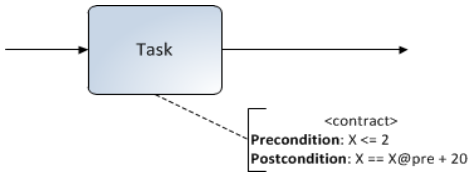

Figure 7. Task annotation

For the task annotation to be written in this way, its specification should be known. However, our work allows tasks that have weak annotations, i.e. annotations that provide little information, e.g."*postCondition: x>0*", and even tasks with no annotation whatsoever, e.g. "*postCondition: true*". In either case the contract could be considered as a partial contract, and the algorithm still works although the results are less accurate than those obtained with a complete contract, since the stronger the contracts, the more useful the test case.

The tasks (services) are provided by various vendors. These suppliers provide the specifications about the tasks in a formal way. If a task has no information related to it, then any unknown behaviour might be performed and any irregularities would therefore remain undetected. This situation negatively affects the test cases of the generation process.

Our approach extracts these annotations from the *ρ* and *P* elements of the *BPC<I, O, ρ, P, Γ>*.

3) Gateways

Gateway annotations are as useful as task annotations. These serve to indicate the path in a formal way. Two alternatives are proposed for the annotation of this information: annotating the links, or annotating the gateway itself. Our approach uses the first option which annotates each link that leaves the gateway. These annotations contain the activation condition expressed as a constraint as shown in Figure 8.

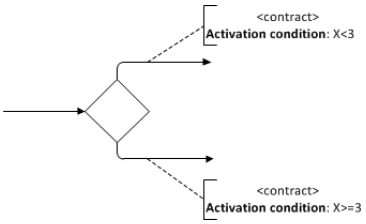

Figure 8. Gateway annotation

The second option certainly appears more complicated than the first alternative since these annotations should contain all the activation information about each branch.

Following the KISS principle (Keep It Short and Simple), derived from the Ockham knife principle, the second option has been selected.

Our approach extracts these annotations from the *Γ* element of the *BPC<I, O, ρ, P, Γ>*.

4) Business Rules

In order to capture the business rules in the *Annotated BP*, an artefact is written. This artefact, represented as a database graphic, is linked to neither task nor gateway,

and defines a set of constraints that hold true throughout the whole process.

The *BR* input is written as a property of this artefact. It could be interpreted as a connection to data base information with the corresponding constraints of the business rules. As mentioned earlier, our approach considers that the *BR* input is a textual set (Θ) with constraints over the variables of the data flow.

These business rules could be applicable to various business processes, and therefore the variables specified in the constraints are not necessarily present in this *BPM*. In our approach, these constraints are not taken into account.

Using these concepts, the *BPM* is annotated. The result of this task, as applied to the example, is shown in Figure 9.

B. *Path Calculation*

Certain papers are focused on the task of path calculation of the graphs [13]. Our approach uses an algorithm derived from an approach described in [10] which focuses on the calculation of the path for a business process, and uses data propagation to achieve this objective.

As the result of the path calculation algorithm, a list of paths is produced. Each path consists of a set of tasks, connected by links, and represents one possible instantiation of the process. Once paths have been calculated, the rest of the process works with each path individually.

The path calculation algorithm takes into account the different types of gateways. The AND gateway executes all the output branches in each test, but the XOR gateway executes only one output branch.

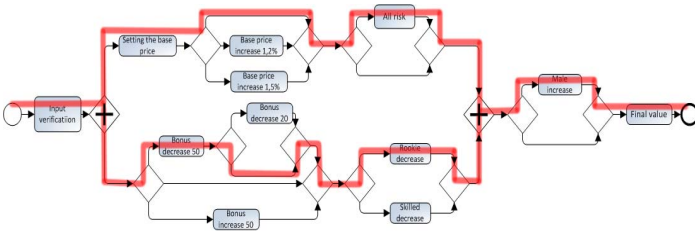

Figure 10. Path #1 calculated

For the example shown in Figure 10, the textual representation would be *<start, input verification, {<Setting the base price, All risk>,<Bonus decrease 50, Novice decrease>}, Male increase, Final Value, end>*.

As shown in Figure 10, the parallel gateway divides the path which is then calculated in two ways. The example shown in Figure 1 has 96 different paths. Some of the paths could be unreachable due to the repeated gateway activation conditions, as occurs with "*horse_power>48"* in the example.

C. *SSA Form Transformation*

Static Single Assignment Form (SSA) is an intermediate representation in which every variable is assigned exactly once. The original variables are split into versions, with new variables typically indicated by the original name with a subscript, so that every definition attains its own version.

SSA Form is equivalent to the original source but

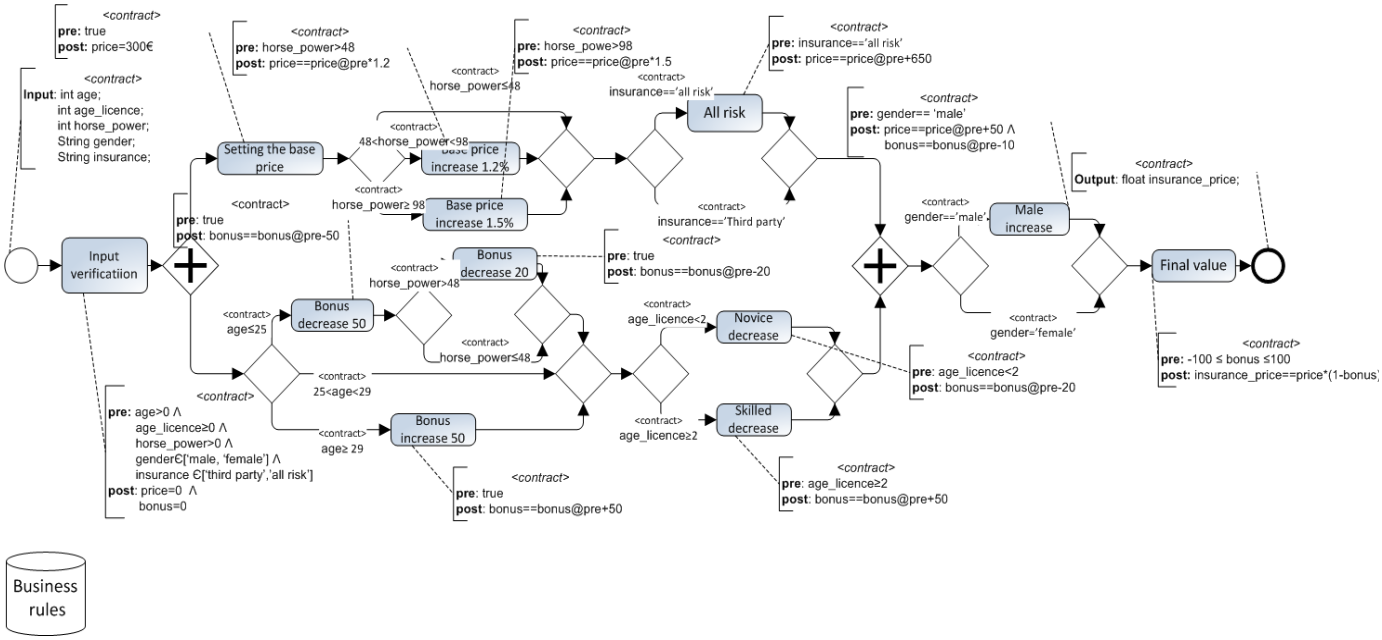

Figure 9. Annotated BPMN model for the motivating example

only allows one assignment for each variable in the trace executed, e.g. for the code *"x = x + 3"*, SSA Form equivalent is "$x_1 = x_0 + 3$". Thus, x_0 maintains the value of the variable *x* before, and x_1 maintains the value of the variable *x* after. The SSA Form transformation preserves the sequence of assigned values, but requires the handling of more variables. For further information see [14].

This transformation is made automatically for each path calculated.

D. Test-Case Data Generation

In this paper, data generation for the test cases is performed based on the constraints paradigm. The test case data generation problem is transformed into various CSPs. A CSP has to be solved for each path calculated in the previous task. Other authors propose an alternative usage of constraints to solve the test case data generation problem [5].

A CSP consists of a set of variables and a set of constraints over these variables. The variables have a range of values. As the constraints are evaluated, these ranges are reduced. After evaluating the whole set of constraints, an assignment of variables within their ranges is returned as the CSP solution. An incoherent set of constraints is detected when any variable is assigned to an empty range. If an incoherency is detected, then the contract of the business process is also deemed incoherent. In Figure 11 an example of incoherence is shown.

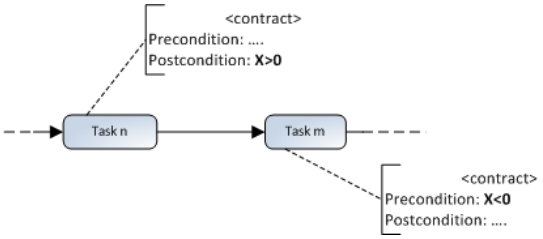

Figure 11. Incoherent set of contraints

Each CSP has a different set of variables and constraints. The constraints are those that appear in the path associated to the CSP and also those that appear in the business rules.

Our approach uses a tool called COMET [15] to solve the CSPs. Each solved CSP provides an instance of test case data. Test case data consists of values assigned to each input and output variable. The process is tested with a test case generated from an instance of test case data. As shown in Figure 5, if the output value is the same as that expected, then the test case associated to this test case data is considered as "*pass*", otherwise the test case is considered as "*fail*".

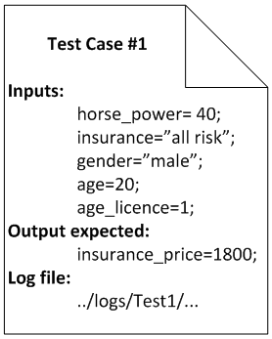

Figure 12. Test Case Data for path #1

The result of the Test Case Data Generation task is a list of test case data. One instance of test case data is shown in Figure 12. Each instance of test case data corresponds to a path in the model. Our approach is focused on obtaining test cases in order to achieve a given objective. These test cases make up a test suite. If all the test cases in a test suite are passed, then the objective is considered as achieved; otherwise the objective remains unaccomplished. A test suite that executes all the tasks in the *BPM* must then be generated for the "*Complete Cover*" objective.

Each instance of test case data has an identification value, and hence a test suite is represented as a tuple $<\{id_1, \ldots, id_n\}, Obj>$ with a list of test case data identifications (id_i), and the objective(*Obj)*.

V. Experimental Results

The experimental results are obtained on a core-i7-870 PC with 6GB RAM on Windows 7. The execution time is measured for each task of the *TCDGP* applied to four examples.

XML is used to express the entire information generated by the process. Test case data, paths and contracts are written in XML. The process is integrated into a tool called TREDAR [16], which is developed by the Quivir Research Group [17]. TREDAR provides a BPMN modeller, connects to the COMET tool to solve the CSPs, and also enables the execution of the *BPM*. In order to execute the BPM, the implementation of the tasks must be fully completed.

Java is the language used for the implementation of the first three steps of the TCDGP due to the ease with which TREDAR can integrate Java applications. COMET is applied only in the last step.

To illustrate the behaviour of the *TCDGP* shown in Figure 4, TREDAR is set to externalize all the files calculated in each task. First, the *BPM* is modelled as shown in Figure 13, and TREDAR is also provided with the *BPC* and the *BR*. Both elements are XML files as shown in Figure 14.

TREDAR allows the user to insert a text input to establish the objective to achieve. The inputs *BPM*, *BPC*, *BR* and *Obj* are provided for TREDAR as these are the inputs of the *TCDGP* shown. "*Complete Cover*" is entered as *Obj*.

```
<business_process_contract>
    <inputs>
        <input var="age" type="Integer" />
        <input var="gender" type="String" />
        ...
    </inputs>
    <outputs>
        <output var="insurancePrice" type="Float" />
    </outputs>
    <preconditions>
        <constraint elementID="Base price increase 1.5%"
                    constraint="horsepower > 98"/>
        <constraint elementID="Novice decrease 20" constraint=""/>
        ...
    </preconditions>
    <postconditions>
        <constraint elementID="Base price increase 1.5%"
                    constraint="basePrice *=1.5"/>
        <constraint elementID="Novice decrease 20"
                    constraint="bonus-=20"/>
        ...
    </postconditions>
    <gateways_activations>
        <gateways_activation elementID="Xord 1">
            <constraint elementID="1"
                        constraint="horsepower <= 48"/>
            <constraint elementID="2"
                        constraint="48 < horsepower <98"/>
            ...
        </gateways_activation>
        ...
    </gateways_activations>
</business_process_contract>
```

```
<business_rules>
    <business_rule
        constraint="200<insurancePrice"/>
    <business_rule
        constraint="insurancePrice<200"/>
</business_rules>
```

Figure 13. BPC and BR input files

The result of the first task is the *Annotated BP* which is expressed in JSON format since this format is used by TREDAR to export models. In this *Annotated BP*, annotations are expressed as inner properties. This decision helps towards the efforts to keep the graphical representation as simple as possible. Each constraint is assigned to each element in the process. Therefore, as shown in Figure 4, the first task (Process Annotation) attains the *Annotated BP* output.

```
<paths>
    <path>
        <element elementID="start"/>
        <element elementID="Input verification"/>
        <paths>
            <path>
                <element elementID="Setting the base price"/>
                <element elementID="All risk"/>
            </path>
            <path>
                <element elementID="Bonus decrease 50"/>
                <element elementID="Novice decrease"/>
            </path>
        </paths>
        <element elementID="Male increase"/>
        <element elementID="Final value"/>
        <element elementID="End"/>
    </path>
    <path>
        <element elementID="start"/>
        <element elementID="Input verification"/>
        <paths>
            ...
        </paths>
        ...
    </path>
    ...
</paths>
```

Figure 14. Output file of the Path Calculation Task

The result of the second task (Path Calculation) is shown in Figure 15. This task generates a XML file with all the paths in the *BPM*. In theory, it has to generate

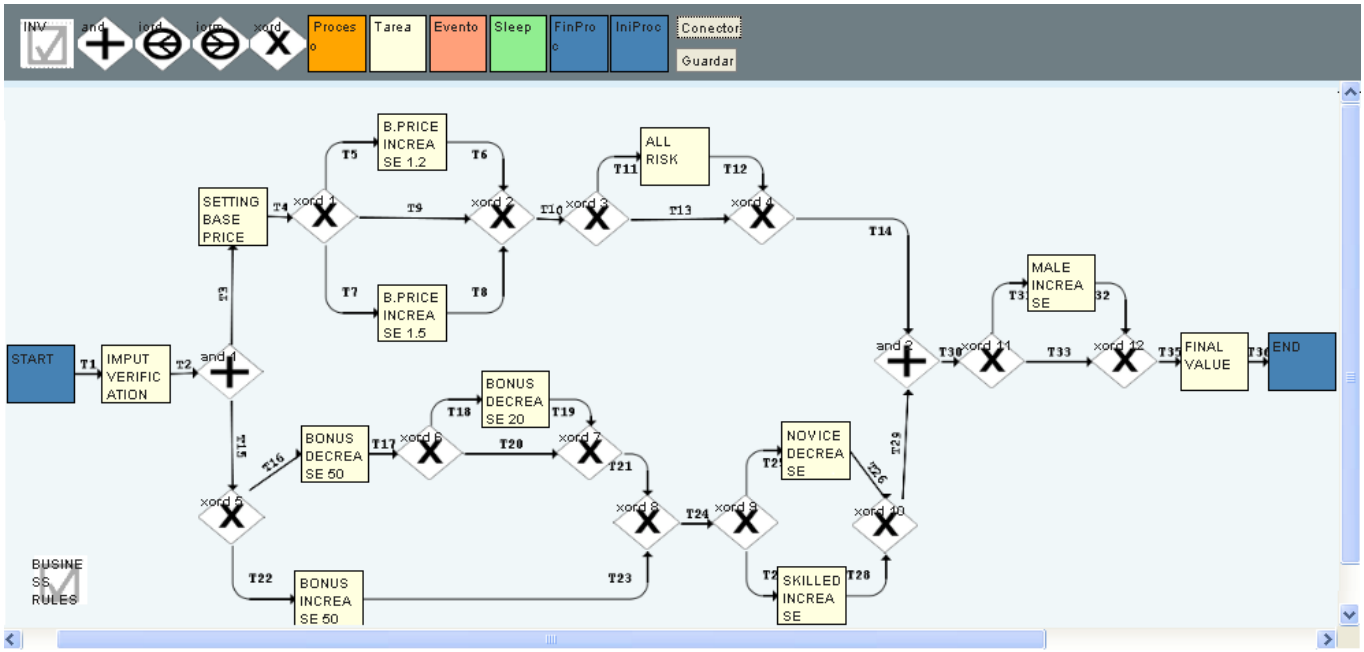

Figure 13. Example in TREDAR

approximately 96 paths. Fewer than 96 paths are generated however, since there are a number of impossible paths due to the presence of the same evaluation conditions at gateways.

The third task transforms each constraint in the path into SSA Form. A modified *Annotated BP* is generated in which the annotations of the elements of the path are transformed.

The last task is the Test Case Data Generation. It generates a XML with different test case data as shown in Figure 16. The sets of test case data are generated depending on the objective selected in the *TCDGP*. In this example, the "*Complete Cover*" objective is selected. This calculates test case data until all the tasks of the *BPM* are covered. If a more in-depth analysis is carried out, only two instances of test case data are necessary. However, since our approach is not focused on the optimization of the results,32 instances of test case data for the "*Complete Cover*" objective are generated. Our approach uses COMET to generate each instance of test case data.

```
<test_case_datas>
    <test_case_data id="1">
        <objectives>
            <objective id="Complete Cover"/>
        </objectives>
        <inputs>
            <input var="age" value="20"/>
            <input var="age_licence" value="1"/>
            <input var="horsepower" value="40"/>
            <input var="insurance" value="'all risk'"/>
            <input var="gender" value="'male'"/>
        </inputs>
        <outputs>
            <input var="insurance_price" expectedValue="1800"/>
        </outputs>
        <logFile path="..."/>
    </test_case_data>
    <test_case_data id="2">
        ...
    </test_case_data>
    ...
</test_case_datas>
```

Figure 15. Test Case Data Task output file

Finally, TREDAR is also employed as a *Test Component* to stimulate the *BPUT*, as shown in Figure 5. Hence, TREDAR executes the process with the inputs indicated in each instance of test case data and the results are compared with the expected output. These evaluations are written in a text file indicated by *LogFile* (this value is predefined to a file in the file system).

A. *Worst-case-scenario execution time*

Further examples are developed in order to study the execution times of each task of the *TCDGP*. These examples are aimed at showing the time elapsed in the worst-case scenario. Three different processes are used with 12, 24, and 36 tasks respectively.

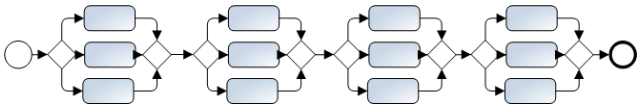

Figure 16. Worst-case scenario for a 12-task process

The worst-case scenario in a *BPM* is achieved by resolving a combinatorial problem. Let *N* be the number of tasks of a business process. How can they be combined in order to create the most complex process (i.e. the process with the most paths)? *N* can be expressed as *m * N/m;* a process with *m* groups of *N/m* tasks combined with a XOR gateway has $m^{N/m}$ paths. The *m* for the worst-case scenario is discovered by calculating the root of the derivate of $x^{1/x}$, i.e. the maximum of the function. The solution is Euler's number '*e*', and since $e \approx 2.7$, the nearest greater integer is 3. Therefore, a process with *N* tasks has, at most, $3^{N/3}$ paths. Hence, the worst-case scenario in a process is achieved by setting the process into groups of 3 tasks, as shown in Figure 17.

Although links with no task may exist in a gateway, this scenario is not taken into account here. This case occurs in the example, Figure 1. These no-tasked links would have to be managed as tasked in order to use the above formula.

The worst-case scenario in a *BPC* is achieved with an infinite number of constraints. However, at least one constraint for each element (task or link of a gateway) is considered in the creation of these examples. Therefore, for a process with *N* tasks and *N* links of the gateways, we write *N* preconditions, *N* postconditions and *N* activation constraints, and hence *3*N* arithmetic constraints are considered. One integer variable for each task is considered, i.e. *N* variables. Since *BR* is affected in the same way as *BPC*, 3 constraints for the *BR* are used in these examples.

A worst-case scenario in an objective is achieved by defining it in order to generate test case data for every path, and hence the "*Complete Cover*" objective is used that generates test data for $2*3^{(N/3)-1}+1$ paths. (This figure depends on the implementation; ours is based on a *Depth-First Search* algorithm).

Test Case Data Generation task solves a CSP for each path: The preconditions of the tasks in a path (*N/3*); the postconditions (*N/3*); the constraints of the gateway links in the path (*N/3*); and the business rules (*3*). Therefore *N + 3* constraints have to be solved for each path.

The three worst-case-scenario processes have been constructed with arithmetic variables and arithmetic constraints. The execution times of each task are shown in Table II. The *TCDGP* has been applied to the example

and to the worst-case-scenario processes with 12, 24 and 36 tasks.

TABLE II. EXECUTION TIMES IN "COMPLETE COVER" OBJECTIVE

	Motiv. examp.	12-task proc.	24-task proc.	36-task proc.
BP Annot.	0.21	0.24	0.35	0.43
Path calc.	0.29	0.4	3.9	280.6
SSA transf.	0.47	0.57	45.91	6257.9
Test gen.	0.7	0.92	63.25	7780.66
TOTAL	**1.67**	**2.13**	**113.41**	**14319.59**

(Values expressed in seconds)

As shown in the table, the last two tasks are the most time-consuming, and hence these are considered as the bottleneck of the *TCDGP*.

The Test Case Data Generation task may vary with alternative objectives. In Table III, execution times are shown for the "*Task 1*" objective that is achieved with only one test. The "*Task 1*" objective is also implemented and it aims to generate a test case that executes, at least, the task identified as "Task 1".

The obtained results may also vary due to the computer configuration of the test and to the existence of other programs being run during the TCDGP. However, results clearly indicate the SSA Form Transformation task as an effective optimization technique.

TABLE III. EXECUTION TIMES IN "TASK 1" OBJECTIVE

	Motiv. examp.	12-task proc.	24-task proc.	36-task proc.
BP Annot.	0.23	0.3	0.34	0.41
Path calc.	0.29	0.35	4.12	310.1
SSA transf.	0.41	0.57	38.8	6129.5
Test gen.	0.4	0.51	0.7	1.05
TOTAL	**1.33**	**1.73**	**33.54**	**6440.75**

(Values expressed in seconds)

VI. CONCLUSIONS AND FUTURE WORK

The objective of this paper is to automatically obtain a test suite for a business process model that has a contract. Our approach provides two main ideas: an annotation-based solution for the inclusion of the contract in the process, and the automatic test case generation for BPMN models from an *Annotated BP* by using constraint programming. The high performance of this paradigm greatly benefits this approach.

This work constitutes a first step towards the generation of test cases in business process models. The main difference with respect to the other approaches is that our work generates test cases against a previously defined contract and that the process is completely automatic.

This approach involves a certain amount of unfinished work and hence various lines of research are proposed:

- First, the BPMN specification is not fully applied in this paper. An extension of this work could be performed in order to cover the complete BPMN specification so that it may be applied to business processes of a more complex nature.
- Second, this work applies SSA Form transformation after the Path Calculation task. This implies that several SSA Form transformations have to be carried out over the same *BPM*, which may lead to a corresponding overload. Since the same variables and constraints may exist in several different paths, SSA Form transformations could produce different results. One possible optimization is the modification over the SSA Form Transformation in order to execute it only once before the Path Calculation task is performed.
- Third, our approach considers each task as a single service with only one operation. A better approach would result if the various operations were taken into account in the same services within the process.
- Finally, the UML Testing Profile should be applied completely, (i.e. each test case must be executed by test components, the verdict (result of a test case) must be analysed by an arbiter, and so on. Future work could involve the application of all the elements of UML Testing Profile in the *TGCP*.

ACKNOWLEDGEMENTS

This work has been partially funded by the Ministry of Economics, Innovation and Science of the Regional Government of Andalusia under grant P08-TIC-04095, and by the Spanish Ministry of Science and Innovation under grant TIN2009-13714, and by FEDER (under the ERDF Programme).

REFERENCES

[1] Dong, W.L, Yu, H., Zhang, Y.B. "Testing BPEL-based Web Service Composition Using High-level Petri Nets." In: Proceedings of EDOC'06, pág. 441-444, IEEE CS (2006). Testing: Academic & Industrial Conference - Practice And Research Techniques (TAIC PART'06). Generation of Conformance Test Suites for Compositions of Web Services Using Model Checking

[2] Jose García-Fanjul, Claudio de la Riva, Javier Tuya, "Generation of Conformance Test Suites for Compositions of Web Services Using Model Checking," Practice And Research Techniques, Testing: Academic & Industrial Conference on, pp.

127-130, Testing: Academic & Industrial Conference - Practice And Research Techniques (TAIC PART'06), 2006.

[3] Didier Buchs, Levi Lucio, and Ang Chen. “Model Checking Techniques for Test Generation from Business Process Models.” Ada-Europe 2009, LNCS 5570, pp. 59–74, 2009. Springer-Verlag

[4] Yan, J., Li, Z., Yuan, Y., Sun, W. Zhang, J. “BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach.” In: Proceedings of ISSRE’06, pág. 75-84, IEEE CS (2006)

[5] Yuan, Y., Li, Z., Sun, W. “A Graph-search Based Approach to BPEL4WS Test Generation.” In: International Conference on Software Engineering Advances (ICSEA 2006), pp. 14. Papeete (Tahiti, French Polynesia) (2006)

[6] Juan José Domínguez-Jiménez, Antonia Estero-Botaro, Antonio García-Domínguez, Inmaculada Medina-Bulo. “GAmera: A Tool for WS-BPEL Composition Testing Using Mutation Analysis.” ICWE 2010: 490-493

[7] Meinke,K. “Automated Black-Box Testing of Functional Correctness Using Function Approximation.” In: ISSTA’04. Proceedings of the 2004ACMSIGSOFTinternational Symposium on Software testing and analysis,pp.143–153.ACMPress,NewYork(2004)

[8] Briand, L.C., Labiche, Y., Sun, H. “Investigating the Use of Analysis Contracts to Support Fault Isolation in Object Oriented Code.” In: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, pp. 70–80. ACM Press, New York (2002)

[9] Object Management Group (OMG). UML Testing Profile Specification, v1.0. Technical report, OMG, 2005.

[10] A. J. Varela- Vaca, Rafael M. Gasca, L. Parody: "OPBUS: Automating Structural Fault Diagnosis for Graphical Models in the Design of Business Processes". 21th International Workshop in Principles of Diagnosis (DX'10). Portland, Oregon, USA.

[11] Pascal Hentenryck. 2009. “Constraint Programming.” In Proceedings of the 5th International Conference on Evolutionary Multi-Criterion Optimization (EMO '09), Matthias Ehrgott, Carlos M. Fonseca, Xavier Gandibleux, Jin-Kao Hao, and Marc Sevaux (Eds.). Springer-Verlag, Berlin, Heidelberg, 3-3. DOI=10.1007/978-3-642-01020-0_3 http://dx.doi.org/10.1007/978-3-642-01020-0_3

[12] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M.Leino, and Erik Poll. “An Overview of ML Tools and Applications.” International Journal on Software Tools for Technology Transfer, February 2005.

[13] Recuero, Alfonso. "Aplicaciones de la teoría de grafos: búsqueda de caminos en una red y análisis de su conectividad" Informes de la Construcción [Online], 46 (1994)

[14] B.Alpern,M.N. Wegman, and F.K. Zadeck. “Detecting equality of variables in programs.” In ACM, editor, POPL ’88. Proceedings of the conference on Principles of programming languages, January 13-15, 1988, San Diego, CA, pages 1-11, New York, NY, USA, 1988, ACM Press.

[15] Comet 2010, [Online] http://www.comet-online.org/

[16] Tredar application 2011, [Online] http://www.lsi.us.es/~quivir/index.php/Tredar/HomePage

[17] Quivir Research Group, [Online]http://www.lsi.us.es/~quivir/

[18] R. Blanco, J. García-Fanjul and J. Tuya. “A first approach to test case generation for BPEL compositions of web services using Scatter Search.” 2nd International Workshop on Search-Based Software Testing. 2009.

[19] Bakota, T., Beszédes, Á., Gergely, T., Gyalai, M., Gyimóthy, T. and Füleki, D. “Semi-Automatic Test Case Generation from Business Process Models.” In Proceedings of the 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering (SPLST 2009 & NW-MODE 2009), pages 5-18. Tampere, Finland, August 26-28, 2009.