Design and Management of Flexible Process Variants using Templates and Rules

Akhil Kumar[1]
Smeal College of Business
The Pennsylvania State University
University Park, PA 16802 USA
akhil@psu.edu

Wen Yao
College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802 USA
wxy119@psu.edu

Abstract. This paper makes four main contributions towards managing large collections of process models. First, we show how flexible process variants can be configured by combining generic process templates and business rules. We configure a customized process variant by applying rules according to specific case data, and running a configuration algorithm. This leads to a separation of control flow and business policy. Secondly, we develop a new succinct representation for process trees as strings by performing a post-order traversal and associating level numbers with each entry. This new structure can facilitate process variant configuration and retrieval. Third, we develop techniques for querying a repository of process variants. The search process is facilitated by means of a bit vectors that capture information about the tasks and control flow structures present in each variant. A variety of queries can be handled with this structure. In this way the management of variants even in a large repository is greatly enhanced. Finally, we present an architecture and describe a preliminary implementation of our approach with a case study for demonstration. Our focus is on capturing deeper process knowledge and achieving a holistic approach to robust process design that encompasses control flow, resources and data, as well as ease of accommodating changes to business policy.

Keywords: flexible process variants**,** rules, templates, configuration, post-order tree traversal, querying

1. Introduction

Among the many approaches and frameworks for designing business workflows, most are based on mapping a control flow that specifies the coordination of various activities (see, for instance, [1-5]). The control flow description of a process is also called a *process schema*. In general, there is a large number of process schemas in an organization. This occurs partly because many schemas are variants of one another with minor differences among them. Take for instance, an insurance company that writes policies for automobile, home and other kinds of insurance. When claim applications are made, the company has to initiate a different process schema for an automobile accident claim as compared to a home damage claim. Moreover, in the home damage claim scenario, a different process must be enacted for a home whose value is less than $100,000 versus a home whose value is more than $250,000. In the former case only one adjuster might be required to visit the home and appraise the damage, while in the second case two adjusters are required to submit independent reports of damage assessment. In general, if there are thousands of *process variants* it makes finding the correct process difficult and error prone.

[1] Corresponding author. Tel: 814-863-0034, Fax: 814-863-7067

Most process management tools do not support configuration and management of process variants [6]. Usually, the above scenarios are handled by two approaches with traditional workflow technologies. The *single-model approach* captures multiple variants in a single process model using conditional branches. Thus it results in a large and complex "spaghetti-like" model, which is difficult to understand and expensive to maintain [6]. In contrast, the *multi-model approach* creates a variant by duplicating a process model and adjusting it to specific needs. This leads to high redundancy because these variants are identical or similar in most part [6]. More complications arise if the company changes its policy to require two independent assessments only when the value of the home is more than $500,000. Just this simple business policy change will necessitate a change in many process variants. It is both time and effort consuming if every variant pertaining to that policy has to be modified manually. *Therefore, business policy should be separated from the process schema.*

Business rules have been used to increase the expressiveness of existing process modeling languages and improve process flexibility [7]. Rules have been used to support run time adaptation of process instances in response to exceptions [8, 9]. Other studies are concerned with improving the design time flexibility of process models. For example, some studies [10-12] have proposed to use rules to model *business logic* that is extracted from *process logic*, leading to a better decoupling of the system. Aspect-oriented process modeling [13-15] also belongs to this stream of research, since it uses "aspects" to modularize cross-cutting concerns shared among several processes. As a result, a process is adaptable at certain points where "cross-cutting" activities are inserted. Although more complexity and effort may arise for managing semantics of rules, rule-based approaches are promising as a useful way to model business logic and improve manageability of large collections of process models.

In this paper we propose a novel solution for process design and management based on the idea of variant configuration. *Process variant configuration* means to generate a process variant on the fly (say, a BPMN [5] or BPEL [16] model) that integrates the control flow, resource needs and data by applying business rules to a generic *process template*, which describes a very basic and general process schema,. In this approach, rules are not only applicable at designated points, but they also allow adaptation of any process pattern in a more general way. This approach offers several advantages. First, it incorporates business policy into process design dynamically. If a policy changes, only the corresponding rules have to be modified while the process template can remain the same [17]. Process variants derived from this process template will be automatically reconfigured to adapt to policy changes without human intervention.

In addition, an end user does not have to create a large number of process schemas beforehand, and manually determine which schema to execute when a case (e.g., a new insurance claim) arrives. Second,

we develop new techniques to facilitate variant search and discovery in a large repository based on representing process schemas as strings. An end user can search a variant based on its process structure using post-order traversal, or retrieve a number of relevant process variants by querying the rule base. We show that our method is very helpful for managing large collections of process models. Third, this approach leads to *holistic process design*. Workflow research has focused on the modeling of the control flow of a process, while other key aspects like data flow and resource needs are neglected. In general, a holistic process model requires additional information like resource needs (e.g., equipment, and facilities) for task completion and data values of parameters associated with a task, etc. Our approach integrates the modeling of resource and data needs of various tasks as well into the process description.

Thus, the essence of our approach is: **process template + rules = process variants**. The rules create specific change operations (e.g., insert or delete a task) based on the actual case data and then apply them to the process template to *generate a variant*. Naturally, this leads to considerably more flexibility than conventional approaches and is suitable in a constantly changing environment, as well as for resource intensive or ad hoc workflows. The organization of this paper is as follows. Section 2 introduces a motivating example, describes formal process representation, and proposes a language for designing process variants. Then, Section 3 deals with rule representation and rule processing. Section 4 describes our configuration algorithm in detail based on the idea of post-order representation of process models. Next, Section 5 discusses process similarity and techniques for searching a large repository of process variants. Section 6 describes our prototype implementation, and is followed by a review of related work in Section 7. Lastly, Section 8 concludes this paper with directions for future work.

2. Preliminaries

2.1. Motivating Example

Figure 1 presents an example process template in BPMN notation [5] for handling an insurance claim. In this template, after a claim is received by a customer representative, it is validated by a clerk to ensure that the customer has a valid policy that relates to this claim. The clerk also makes an assignment to two adjusters who will review and appraise the damage to the auto or the house, and then submit a report. The two adjusters may perform their jobs in parallel, which is indicated by a *parallel gateway* shown as a diamond with a cross. The first parallel gateway splits into multiple branches and has a corresponding parallel gateway where these paths merge. After the reports are received by the customer representative, they are checked for completeness and sent to an officer who will determine the settlement amount. Subsequently, two approvals are required by a manager and a senior manager. Finally, the accounts manager will make the payment to the customer.

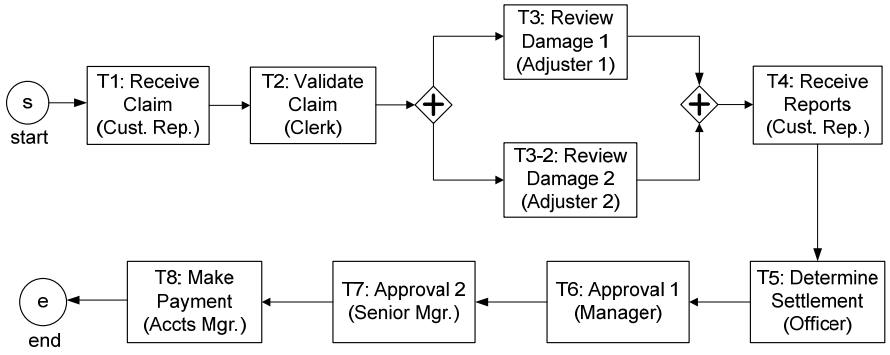


Figure 1. Description of an insurance process in BPMN notation

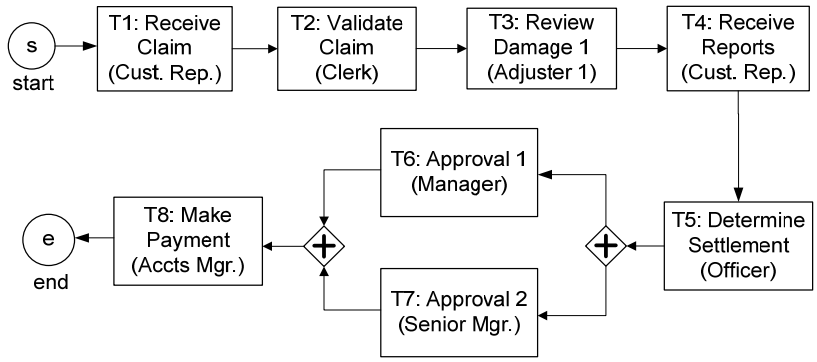
Figure 1 shows the "normal" tasks, the roles that perform them and the control flow relationships among them. However, the actual process in practice may vary depending upon a particular incident or case. Thus, the template should be customized for a specific case by applying rules to it. An example rule set is shown in Figure 2. The rules are written in plain English-like syntax. So, if the loss claimed is less than $500K, then only one adjuster is required (R1); however, if the loss is more than $250K then the adjuster should have more than 10 years of experience (R7) and must fill the "long" form (R9). Further, if the loss is more than $500K, the second adjuster should be classified as an expert (R8). After a settlement is assessed, either one or two approvals are required before payment is made, depending upon the amount of loss (R4, R5, R6). The other two rules deal with the urgency status of the case. If it is marked *expedite,* then the approvals may be performed in parallel to save time (R2). On the other hand, if it is marked *urgent* and the loss is small, then the second approval is deferred until payment is made (R3).

R1: If loss < $500K, **then** skip review by adjuster 2
R2: If application = expedite **then** perform approvals in parallel
R3: If application = urgent and loss < $500K **then**
defer second approval until after payment
R4: If loss < $100K, **then** need manager approval
R5: If $100K < loss < $500K, **then** need manager & senior manager approval
R6: If loss > $500K, **then** need manager + VP approval
R7: If loss > $250K, **then** need adjuster with minimum 10 years experience
R8: If loss > $500K, **then** need detailed assessment from an expert
R9: If loss > $250K, **then** adjustor must fill the "long" form

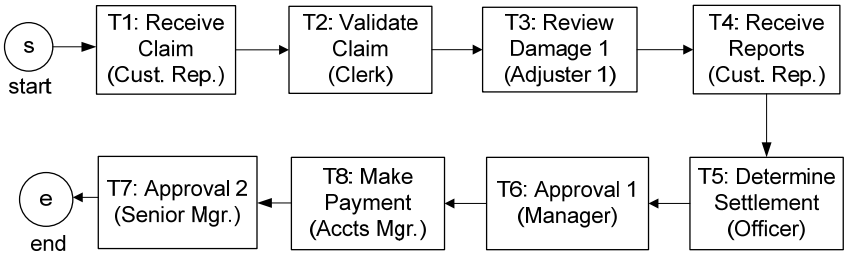
Figure 2. Rules to be applied to the process template of Figure 1

Clearly, by applying rules to the template based on different case data, we obtain different process variants. For example, Figure 3 shows two variants V1 and V2, derived from the process template in Figure 1. In part (a), the loss is $200 K and it is marked *expedite*, while in part (b) the loss is $300K and it is marked *urgent*. We can see that these two variants are identical in most parts and only have minor

difference in policy. The full process that captures all the variants can be very large and complex, as shown in Appendix 1. We will evaluate its complexity later in Section 5.



(a) Process variant V1: (loss = $200K, status = expedite)



(b) Process variant V2: (loss = $300K, status = urgent)

Figure 3. Two process variants from process template and rules

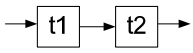
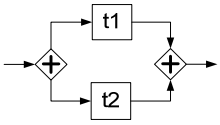
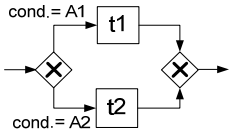
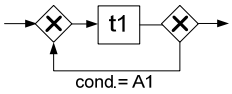
Informally, the basic criteria for differentiating business logic (or business rules) from process logic is that rules are more dynamic (e.g., approval procedure is different depending upon the damage and the urgency status), while process logic is relatively static and hardly changes over time. The purpose of these examples was to motivate the need for configuring process variants by separating business policy from process scheme. *Our motivation in a nutshell is to reduce the redundancy between process models that are slightly different from each other, to make process models more adaptive to business policy changes, and to efficiently retrieve the desired model for a specific case.* Thus, our approach uses process templates to abstract process logic for similar cases, and rules to better handle dynamic requirements.

2.2. Formal Representation of a Business Process Model

In general, a business process can be composed by combining four *basic constructs* as building blocks, i.e. *sequence*, *parallel*, *decision structure* (or *choice*) and *loop*, as shown in Table 1. They can be applied to two or more atomic tasks, e.g. S(A,B, C), or P(A,B,C), to indicate that tasks (or subprocesses) A, B and C are in sequence, or in parallel, respectively. Parallelism is introduced by using a parallel split gateway to create two or more parallel branches which are synchronized by another parallel merge gateway. We

use 'P' to denote this structure. Similarly, a choice structure, denoted as 'C', is created with a pair of *exclusive OR (XOR) gateways*. The first XOR node represents a choice or a decision point, where there is one incoming branch and two or more outgoing branches, exactly one of which can be activated. Finally, a loop, denoted by 'L', is also drawn using a pair of XOR gateways but differently from a choice structure. The first XOR gateway takes only one out of all the incoming branches and the second XOR gateway represents a decision point that can activate any one of the outgoing branches.

Table 1. Basic patterns to design processes

Structure	**Graphical Representation (BPMN)**	**BPEL Representation**	**Notation**
Sequence		<sequence name= "S1"> <invoke name="t1"/> <invoke name="t2"/> </sequence>	S(t1, t2)
Parallel		<flow name="P1"> <invoke name="t1"/> <invoke name="t2"/> </flow>	P(t1,t2)
Choice		<switch name="C1"> <case condition="A1"> <invoke name="t1"/> <case/> <otherwise name="A2"> <invoke name="t2"> </otherwise> </switch>	C(t1, t2)
Loop		<while name="L"> <condition name="A1"> <invoke name="t1"> </condition> </while>	L(t1)

The patterns are applied recursively to create complex processes. In our paper, we use BPEL [16] to represent process schema since it is one of the most popular process modeling languages. Table 1 also describes how BPEL is mapped into BPMN notation. Figure 4 shows the BPEL representation of the insurance claim process in Figure 1 (the data and resources pertaining to this process are omitted). In general any structured process can be represented in this way. Other process features, such as events and structured activities, can also be expressed in BPEL.

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="insurance process model">
<variables>
     <variable name="loss" type=″xsd:double″/>
     <variable name="status" type:″xsd:string″/>
</variables>
<sequence name="S1">
     <invoke name="Receive Claim"/>
     <invoke name="Validate Claim"/>
     <flow name="P">
            <invoke name="Review Damage 1"/>
            <invoke name="Review Damage 2"/>
     </flow>
     <invoke name="Receive Reports"/>
     <invoke name="Determine settlement"/>
     <invoke name="Approval 1"/>
     <invoke name="Approval 2"/>
     <invoke name="Make payment"/>
</sequence>
</process>
```

Figure 4. BPEL representation of the insurance process model in Figure 1

2.3. Language to Design Flexible Process Variants (FlexVar)

A *process template* captures the common process logic among a family of process models and serves as the basis for deriving variants. Thus, it can also be called a *base process*. A *process variant* is *configured* from a process template by applying a small number of change operations to it through rules. Strictly, the process template is also another variant; however, it is chosen in such a way that it has the closest degree of similarity to other variants in a collection of related processes. The process of configuring a variant from a process template by applying a series of change operations is called *variant configuration*.

Table 2 presents a simple language called **FlexVar** consisting of basic change operations that can be used to modify a process template. In general, these operations can be categorized into three types that correspond to the *control flow, resource,* and *data related* perspectives of process modeling. Control flow related operations can change the execution sequence of activities in a process template, e.g., *insert, delete* or *move* a task; resource related operations can change the role that performs a particular task or activity, e.g., assign a different performer for the task; data related operations can modify the properties of activities, resources, and other process metadata. With these operations, we can design a process variant from different perspectives. For example, the "insert" operation allows us to insert a new task into a process template. However, we must also specify the control-flow relationship of the new task with an existing task, i.e., in a *sequence, parallel, choice,* or *loop* structure. For a sequence, the user should specify whether the task is inserted "before" (S_b) or "after" (S_a) another task in the process. This would also apply to a loop structure to indicate the flow sequence within the loop. However, it is not required for "parallel" and "choice" constructs.

Note that in Table 2, the various operations can just as equally apply to *process fragments* (or *subprocesses*) as to tasks. An example of a process fragment in Figure 3(a) is the parallel structure formed by tasks T3 and T3-2. In general, a fragment *f* can similarly be deleted, inserted, etc., as a regular task.

Table 2. Base operations of FlexVar language for modifying a process template

Perspective	**Base operations**	**Description**
Control flow related	delete(*t*|*f**)	Delete task *t*|*f* from the process
	insert(*t*|*f*, S_b|S_a|P|C|L_b|L_a, *t1*, [*N*])	Insert task *t*|*f* in sequence, parallel, choice or loop with task *t1* to create a node *N* (optional) (S_b |L_b = before; S_a |L_a = after)
	replace(*t1*|*f1*, *t2*|*f2*)	Replace *t1*|*f1* with *t2*|*f2*
	move(*t*|*f*, S_b|S_a|P|C|L_b|L_a, *t1*|*f1*)	Move *t*|*f* to a different position. The new place is defined in relation to *t1*|*f1*.
	change(*t1*|*f1*, *t2*|*f2*, S_b|S_a|P|C|L_b|L_a)	Change relationship between *t1*|*f1* and *t2*|*f2* to S_b, S_a, P, C, L_b, or L_a
Resource related	role(*t*, *r*)	Task *t* is performed by role *r*
Data related	data(*attribute*, *value*)	Assign a value to a data attribute
	prop(*role*, *property_name*, *value*)	Each role can have several properties and corresponding values with them
	data_in(*t1*, *din*)	*din* is an input data parameter for task *t1*
	data_out(*t2*, *dout*)	*dout* is an output data parameter for task *t2*
	task (*t*, *name*)	Assign a new name to the task with id *t*
	status(*proc_id*, *value*)	Change the status of a process (value = *normal*, *expedite*, *urgent*, *OFF*)

*Note: *f* represents a process fragment or a subprocess

3. Rule Representation and Processing

3.1. Rule Representation

We use a formal language to represent rules that can be easily written and recognized by a process designer. The format of the rule language is presented as follows:

Rule *rule_name*, *rule_group*, *rule_category*, *priority*
    IF *condition 1*, *condition 2* … *condition n*
    THEN *action 1*, *action 2* … *action n*
END

Here, the parameter *rule_name* assigns a unique name to each rule. A *rule_group* clusters rules with similar functionalities so they can be triggered or disabled at the same time, *rule_category* categorizes type of rules according to process perspectives (see more details in Section 3.2), and *priority* specifies the order in which the rule is triggered. If two rules are both triggered and they conflict with each other, the one with higher priority will be executed.

Once conditions are met, the corresponding rule is triggered and produces actions. Conditions can be connected by logical operators including AND, OR, and NOT, to define complex conditions. An action can be any change operations presented in Table 2 to modify a process template. For example, we can use this rule language to rewrite R3 (see Figure 2) so it can be processed by a rule engine as follows:

Rule R3, RG1, Control_flow, 10
IF process.status = "urgent" AND process.loss < 500000
THEN move (t7, S_a, t8)
END

3.2. Rule Categories

In general, the actions carried out by rules can affect the control flow, resource, and data perspectives of a process. Thus, we categorize rules in terms of their effects as:

- Control flow related rules are used to alter the control flow of a process based on case data. In addition to deleting or replacing a task, they can also alter the control flow by moving a task to a different position, or changing the relationship between two tasks.
- Resource related rules are concerned with resource assignment based on case data.
- Data related rules are associated with properties or attributes of a resource related to a case.
- Hybrid rules concern the modification of *several aspects of process design*. For example, they might alter the flow of control of a process as well as change the properties of a resource.

Control flow related

```
R1: process.loss < 500000 → delete(t3-2)
R2: process.status=”expedite” → change(t6,t7,P)
R3: process.status=”urgent” AND process.loss < 500000 → move(t7, Sa, t8)
```

Resource related

```
R5: process.loss > 100000 AND process.loss < 500000 → role(t7, senior_manager)
R6: process.loss > 500000 → role(t7, vice_president)
```

Data related

```
R7: process.loss > 250000 → prop(adjuster2, min_exp, 10)
R8: process.loss > 500000 → prop(adjuster2, qualification, expert)
R9: process.loss > 250000 → prop(form, type, long)
```

Hybrid Rules

```
R4: process.loss > 0 AND process.loss < 100000 → role(t6, manager) AND delete(t7)
```

Figure 5. Formal representation of business rules in Figure 2

In addition, a hybrid rule may affect a process template in more than one perspective. These categories can help to organize rules and define them systematically. For simplicity, we use the format **Rule_name: conditions → actions,** where the conditions appear on the left hand side and the actions on the right. Figure 5 shows how the rules in Figure 2 are expressed in this format. They refer to the process template in Figure 1, and the actions are based on the operations presented in Table 2.

Rules R1, R2 and R3 concern the control flow perspective. For example, R1 deletes task t3-2 from the process template if the amount of loss is less than $500K since the review by the second adjuster is not required. R2 changes the relationship between tasks t6 and t7 from "sequence" to "parallel" if it is marked *expedite*. Similarly, R3 moves the second approval task t7 to the end of the process if the case is *urgent*. Rules R5 and R6 are resource related rules and they make assignments of resources to tasks based on case data, i.e., the amount of loss. Rules R7, R8 and R9 are related to properties of the resources, etc in the process. Finally, Rule R4 is a hybrid rule since it not only changes the control flow but also resources of the insurance process.

3.3. Rule Processing and Semantics for Conflict Resolution

When the above rule set is fed with specific case data, corresponding rules are triggered if their condition clauses are satisfied. Then the resulting change operations are applied to the process template to configure a variant. Assume the case data is as follows:

```
loss = $300k; status = expedite.
```

Now, the rules triggered are: R1, R2, R5, R7 and R9. The corresponding operations/actions that become true as a result of applying these rules are:

```
Operation 1: delete(t3-2)
Operation 2: change(t6,t7,P)
Operation 3: role(t7, senior_manager)
Operation 4: prop(adjuster, min_exp, 10)
Operation 5: prop(form,type,long)
```

These operations can be applied to the process template to produce a variant for this specific case or scenario. Although all these rules have been validated by domain experts, they may produce *conflicting results depending upon the order in which they are fired*. For example, insert(t2, S_a, t1) and insert(t3, S_a, t1) applied to a process consisting of a single task 't1' can result in two process fragments S(t1, t2, t3) or S(t1, t3, t2) depending upon the order in which they are applied. Moreover, sometimes rules may fail. For instance, insert(t1, S_a, t2) would fail if task 't2' has been already deleted by an operation in the previous step. Therefore, it is very important to specify the correct semantics for handling such situations for variant configuration. Some possible semantics are:

- Arbitrary semantics: does not impose any order on the rules. This implies that all execution orders of rules are acceptable.
- Priority semantics: assigns a priority to rules if the execution order is important. Higher priority rules will execute first, followed by lower priority one in descending order.
- Complexity semantics: defines a complexity factor to each rule according to the number of conditions. A rule with more conditions is more specific than a rule with fewer ones and is execute first.
- Recency semantics: assigns a time factor to each rule when it is inserted into the rulebase. The more recent rule is activated with a higher priority.
- Fail semantics: returns failure or throws an exception when a rule cannot be executed due to different reasons. In this case a variant cannot be generated. Therefore, the user will have to intervene to modify the rules or assign new priorities to them.

In general, a user should specify the semantics depending on the application. Otherwise, arbitrary semantics will be used by default. If the configuration fails, an exception is generated and the user is notified about the conflicting rules that cause it.

3.4. Checking validity of a sequence of change operations

Before a series of change operations can be applied to create a variant, one must ensure that they are compatible and the resulting variant is structurally correct (e.g., it does contain an infinite loop, etc.). The matrix in Table 3 shows how conflicts between a pair of change operations can be identified to resolve this issue. Assume that operation *op1* is applied first, followed by operation *op2*. There are three resulting cases: (1) *op1* and *op2* are compatible and should be performed independently, e.g. insert (*new*, X, *old*) and insert (*new'*, X', *old'*); (2) *op1* and *op2* are compatible and these two operations can sometimes be merged to produce no effect. For example, the operations insert (*new*, X, *old*) and delete (*old'*) will have no effect if *old'* equals *new*; (3) *op1* and *op2* are incompatible and users should be notified, e.g., delete (*old*) and insert (*new'*, X', *old'*) if *old'* equals *old*. Further, if, say, a task *t* is deleted and reinserted in the same position the net result will be move (t, X, t), which implies no change. Next, we discuss the variant configuration algorithm.

Table 3. A matrix for verifying the correctness of multiple operations

op1 \ op2	**insert (*new'*, X', *old'*)**	**delete (*old'*)**	**Replace (*old'*, *new'*)**	**move (*old1'*, X', *old2'*)**	**change (*old1'*, *old2'*, X')**
insert (*new*, X, *old*)	**compatible**	If *old'* = *new*, then do nothing*	If *old'* = *new*, then insert (*new'*, X, *old*)	If *old1'* = *new*, then insert (*new*, X, *old2'*)	compatible
delete (*old*) / **replace (*old*, *new*)**	If *old* = *old'*, then op1 and op2 are **incompatible**			If *old* = *old1' or old=old2'*, then op1 and op2 are **incompatible**	
move (*old1*, X, *old2*)	**compatible**	If *old'* = *old1*, then skip op1; If *old'* = *old2*, then replace (*old2*, *old1*)	If *old'* = *old1*, then move (*new'*, X, *old2*)	If *old1'* = *old1* and *old2'* = *old2*, then skip op1	If *old1'* = *old1* and *old2'* = *old2*, then move (*old1*, X', *old2*)
change (*old1*, *old2*, X)	**compatible**	If *old'* = *old1* or *old'* = *old2*, then skip op1	compatible	compatible	If *old1'* = *old1* and *old2'* = *old2*, then skip op1

Note: X=$S_b|S_a|P|C|L_b|L_a$; *old* represents existing tasks and *new* represents new tasks, resp.; "=" checks equality
*Note: Otherwise, it is compatible. Same applies to other results in this table.

4. Process Variant Configuration and Representation

The rule processing stage generates a list of change operations related to control flow, resources, and data of a process. The operations that relate to resources (such as role operations) and also those related to data (such as data or prop operation) are added to the case database directly. For example, prop(form,type,long) assigns the value long to the type attribute of the form object. Such data would be used as input data for task execution. However, the operations that relate to the control flow are used as input for a variant configuration algorithm in order to generate a new control flow schema for that variant.

4.1. Overview

Our variant configuration algorithm is based on creating a tree for a process template, and then applying each change operation to the tree. After all the changes are applied, the resulting tree reflects the control flow of the variant. Although, this can be converted into any process description language to describe the process schema, here we use BPEL. Figure 6 shows the process tree of the template in Figure 1, and Figure 7 presents our variant configuration algorithm. There are several equivalent representations of such a tree. *However, this is not the focus of our study*, since it does not make a difference in the configuration process. In general, there are two types of nodes in a process tree: the leaf nodes are the *task nodes*, while the internal nodes are *control nodes* that capture the relationships (e.g. S, P, C, or L) among tasks and/or process fragments. The child nodes of a sequence node are numbered in order from left to right. Thus, the leftmost child appears first in the sequence, and the rightmost one last. For parallel and choice nodes, the order of appearance of the child nodes does not matter because of their execution semantics. In general, a

loop node could have two child nodes, the first for the forward path and the second for the reverse; hence, the order does matter. This tree can be stored in a tabular data structure as follows:

(node_id, type, child_node, in_order),

where

node_id is the ID of the tree node
type refers to four control flow relations (i.e., S, P, C, or L)
child_node is the set of all the child nodes of the current node
in_order is a Boolean value (Y/N) to indicate if the order of child nodes matters as in 'S' and 'L' relation

For example, node S1 is used to connect two control nodes and can be represented as (S1, S, {S2, S3}, Y). In contrast, node P1 is a parallel control node and it is used to connect two tasks. It can be represented as (P1, P, {T3, T3-2}, N).

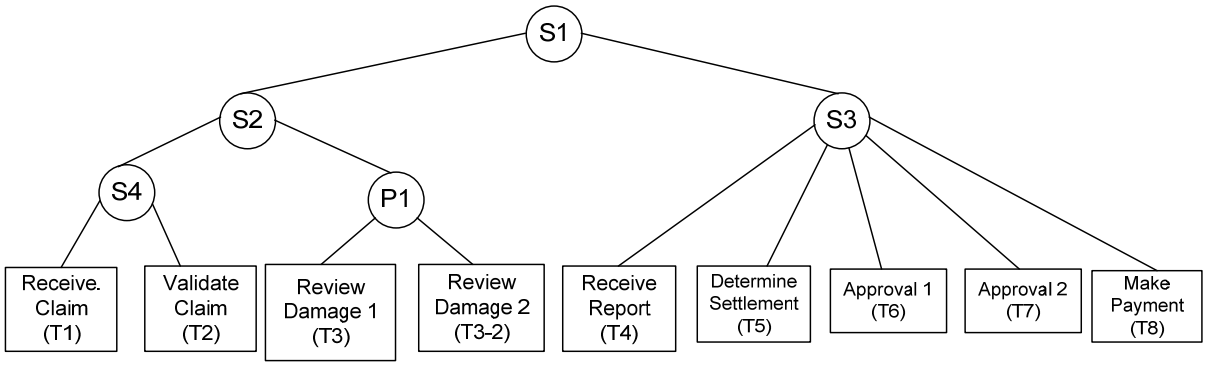


Figure 6. A process tree derived from the template in Figure 1

Algorithm Variant_Configuration (*PT*, *case_data*)
PT: the process template in BPEL
case_data: the initial case data

```
1.  Generate a series of change operations op_set based on case_data and rule set  //run rule engine
2.  Transform PT in BPEL format into a process tree P_TREE
3.      FOR each operation op in op_set //modify the process tree by executing all change operations
4.          Apply op to P_TREE using the configuration algorithm description in Table 4
5.          If operation failed
6.              Throw Exception
7.          Else
8.              Save P_TREE   //   the change is applied successfully
9.      End FOR
10. Transform P_TREE into P_VARIANT in BPEL format
11. Return process variant P_VARIANT
12. END
```

Figure 7. Variant configuration algorithm

4.2. Algorithm Details for Tree Operations

Table 4 shows the detailed pseudo-code for implementing the control flow change operations that were introduced in Table 2. We illustrate these operations next in the context of Figure 6. Although, this tree only presents the control flow perspective of the process, the resource and data related information can also be included in the nodes. A **delete** operation simply removes the node corresponding to a task in the tree, and if the parent of the deleted node has only one child left, then the child is moved up to take the place of the parent. Each non-leaf node should have at least two child nodes and a task is always a leaf node. For simplicity, we discuss operations on task nodes (i.e. at the leaf level) only here. Later we will extend this approach to perform operations on internal nodes.

When a task is to be **inserted,** its position must be specified in the tree with respect to an already existing task node. Moreover, the relationship between the existing node and a new node should also be specified as sequence (S), parallel (P), choice (C) or loop (L). If it is a sequence (or a loop) then it is also necessary to state whether the new task is inserted before (S_b) or after (S_a) the current node. The insert procedure is to create a parent node P1 for the existing node (say, t1) and insert the new node t as a child of P1 in the tree. The parent node can optionally be given a new label N. The **replace** operation simply changes the label of a task node with its new name.

Table 4. Details of performing change operations on a process tree *P_tree*

Operation	**Algorithm description**
Delete (Node *t*)	IF parent (*t*) has $\leq$ 2 child nodes Replace(*parent(t)*, *t.sibling)*; ELSE Delete (*t*); [Exception: Node t is not *P_tree*]
Insert (Node *t*, Rel *X*, Node *t1*, [*N*]) Note: X = S_b|S_a|P|C|L_b|L_a	Create a new parent node *N* for *t1* && *N.node_type=X*; Add *t* as a new child of *N* ;} [Exception: Node *t1* is not in *P_tree*]
Replace (Node *t1*, Node *t2*)	Rename node *t1* with *t2*; [Exception: Node *t1* is not in *P_tree*]
Move (Node *t*, Rel *X*, Node *t1*)	Delete (*t*) && Insert (*t*, *X*, *t1*); [Exception: Node *t1* is not in *P_tree*]
Change (Node *t1*, Node *t2*, Rel *X*)	Change parent (*t1*, *t2*).*node_type* to new relationship *X*;} [Exception: Nodes *t1*,*t2* not in P_*tree*]

The **move** operation is like a delete followed by an insert. It removes a task from its current location in the tree and inserts it into a new position. This new position is defined with respect to an existing task node in the tree which serves as an anchor node. The **change** operation may be used to modify the

relationship between two existing nodes t1 and t2 in the tree. However, this is possible only if a direct relationship (i.e., with a common parent node and no other siblings) exists between them. Otherwise, the operation would fail. In order to implement this operation, we first check if a direct relationship either exists already or can be found by rewriting the tree into an equivalent tree by means of rewriting rules. If it is possible to do so, then the parent node of t1 and t2 is changed to the new relationship. Otherwise, an exception is generated.

These operations are summarized in Table 4. As an example, Figure 8 is derived from Figure 6 by performing delete (T3-2) and change (T6, T7, P) operations.

We next show how a process tree is represented as a text string for ease of storing and searching, and also how the various operations discussed above are translated into search and replace operations on the string to transform one variant into another.

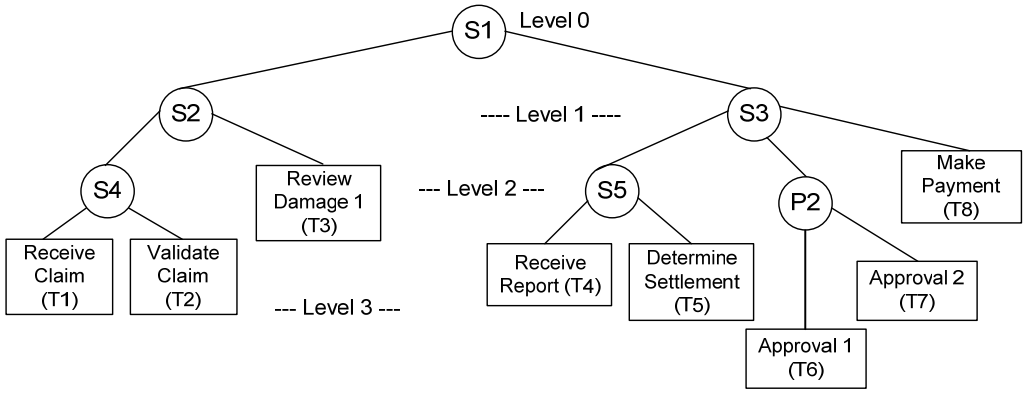


Figure 8. Revised process tree after applying the variant configuration algorithm

4.3. Tree Traversal, Representation and String Operations

We developed a novel representation structure for storing a process variant by traversing all the nodes of its tree in *post-order*. This traversal scheme is: *starting at the root node of the tree, recursively explore its left child sub-tree first, then its right child sub-tree and finally the root itself.* A post-order traversal of the tree in Figure 8 is shown in Figure 9 (a). Another representation for this structure is shown in Figure 9 (b). An alternative representation as two equal length TV and LV vectors is shown in Figure 9 (c). Note that these two representations are equivalent. This figure also shows a derived delta vector, i.e. the difference between successive level numbers in the string, the delta for position i in the string is level[i] – level[i-1]; for position 0 it is 0.

T1-T2-S4-T3-S2-T4-T5-S5-T6-T7-P2-T8-S3-S1

(a) A standard post-order traversal of the tree in Figure 8

T1(3)-T2(3)-S4(2)-T3(2)-S2(1)-T4(3)-T5(3)-S5(2)-T6(3)-T7(3)-P2(2)-T8(2)-S3(1)-S1(0)

(b) A post-order traversal of the tree in Figure 8 with level numbers in parentheses (root is level 0)

Task vector (TV)	T1	T2	S4	T3	S2	T4	T5	S5	T6	T7	P2	T8	S3	S1
Level Vector (LV)	3	3	2	2	1	3	3	2	3	3	2	2	1	0
Delta	0	0	-1	0	-1	2	0	-1	1	0	-1	-1	-1	-1

(c) Alternative representation of post-order traversal as TV, LV and a derived delta vector

Lemma 1. A post-order traversal with levels is equivalent to the tree representation and vice-versa.

Proof sketch. Given a post-order traversal with levels we can scan the entries one at a time and add them to a tree. Further, given a tree we can first add levels to the nodes with the root node at level 0. Then we can create a post-order traversal using the standard algorithm described above. ■

This representation serves two purposes: (1) it simplifies the performance of various operations such as insert, delete, replace on this tree as string operations; and, (2) it facilitates the storing of a process succinctly and querying it. We consider each aspect in detail next.

Various operations on the tree can be directly performed as search and replace string manipulation operations on the post-order traversal string (POS). The details for each operation are shown in Table 5. For example, a *delete* node operation, simply deletes a single node entry from POS if the node is a task node. If it is an internal node, then the subtree rooted at that node must be deleted. For example, if we wish to delete node P2 in Figure 8, the subtree with P2 as its root is represented in post-order as the string: "T6(4)-T7(4)-P2(3)." To delete this subtree, we remove the entry P2, and all contiguous prior entries with level numbers higher than the level number of P2. Therefore, T6 and T7 will also be deleted.

A *reorganize* step follows the deletion of a node. In this step, we check the delta value of a deleted node's neighbor (an adjacent node at the same level as the deleted node). If the neighbor has a non-zero delta, then it means that the deleted node did not have a "sibling". Therefore, the parent of the deleted node is also deleted, while the level of the sibling (and any children of the sibling) is reduced by 1. An *insert* operation may insert a single task, or even a subprocess. In the former case the task is a single

element; in the latter case the subprocess is represented as a post-order string. In both cases a string replacement operation is performed as shown in the table. The *move* operation consists of two parts. The first part is like a delete where a contiguous substring is removed from POS. The second part is like an insert where the deleted string is inserted in a new position in POS. The *change* operation modifies the relationship between two task nodes or, in general, two subtrees. To do this we find the first common ancestor of the two nodes and change its value to the new relationship.

Table 5. Steps of operations to be performed on post-order traversal string (POS)

Operation	**Detailed Steps**
Delete(**n**)	Remove node **n** and higher level numbers to the left until a lower level reached; Reorganize(n)

Insert (**n**, **X**, *n1*)
Note:
X = S_b|S_a|P|C|L_b|L_a
	Replace ("n1", "n1 **n X**", POS); or Replace ("n1", "n1 **n X**", POS);
	level#(X) = level# (n);
	level# (n1)= level# (n) +1; level# (n)= level# (n) +1;
	Renumber child nodes of n with respect to number of n

Move(**n**, X, n1)	**SP1** = Remove node **n** and higher level numbers to the left until a lower level reached; Reorganize(n); Insert(SP1, X, n1)
Change(**n1**,**n2**,X)	Scan substring of POS within entries **n1** and **n2**; Find the highest level control node in the substring; Replace node with X
Reorganize(**n**)	If a *right or left neighbor NN of n* has a **non-zero delta** with pred. **and** succ. nodes, **then** (1) Delete the *neighbor of NN* with the lower level number (there will be only one such neighbor) (2) Decrement level number of node NN (3) Decrement level number all nodes in (position(NN)– i) s.t. Level#(position(NN)– i) > Level(NN) [Note: a neighbor of **n** is an adjacent node to **n** in POS, and at the same level as **n**]

4.4. Examples

Example 1: Figure 10 shows various operations on the running example of Figure 9. It gives the initial POS string (or vector), along with the corresponding level number and delta values. It also shows the effect on each of these values after deleting task T3 and reorganizing; then deleting T5 and reorganizing; and finally inserting T9.

Example 2: In general a variant V_i is transformed into another variant V_j by a series of operations op1, op2, … (e.g. delete, insert, change, etc.) on V_i. Consider the two variants V1 and V2 shown in Figure 11. V2 is derived from V1 by performing the six operations shown in the figure. Table 6 shows the POS and level# vectors for V1 in the first row. In successive rows it shows the effect of each operation on these vectors. The last row shows that the vectors for variant V2 are the same as the resulting vector in the row above except for only a minor structural difference related to T1-T3-T2-T4 task sequence. Note that in a process diagram a sequence is shown by a flow using directed arrows without a special symbol. However,

in a tree because there is no notion of flow, a node is introduced corresponding to the flows of S1, S2, etc. in the example of Figure 11.

	Operation	Post-order traversal string (POS) with levels													
TV0 LV0		T1 3	T2 3	S4 2	T3 2	S2 1	T4 3	T5 3	S5 2	T6 3	T7 3	P2 2	T8 2	S3 1	S1 0
Delta		0	0	-1	0	-1	2	0	-1	1	0	-1	0	-1	-1
	Delete T3; calculate delta	T1 3 0	T2 3 0	S4 2 -1	X	S2 1 -1	T4 3 2	T5 3 0	S5 2 -1	T6 3 1	T7 3 0	P2 2 -1	T8 2 0	S3 1 -1	S1 0 -1
	Reorganize:S2, S4 both have non-zero delta → -Delete S2 -decrement S4,T2,T1 levels	T1 2 0	T2 2 0	S4 1 -1	T4 3 2	T5 3 0	S5 2 -1	T6 3 1	T7 3 0	P2 2 -1	T8 2 0	S3 1 -1	S1 0 -1		
	Delete T5; Reorganize	T1 2 0	T2 2 0	S4 1 -1	T4 3 2	T6 3 0	T7 3 0	P2 2 -1	T8 2 0	S3 1 -1	S1 0 -1				
	Insert(T9, S4, P)	T1 2 0	T2 2 0	S4 1 -1	T9 2 1	P3 2 0	T4 3 1	T6 3 0	T7 3 0	P2 2 -1	T8 2 0	S3 1 -1	S1 0 -1		

Figure 10. Illustration of operations on the POS string of the process in Figure 8

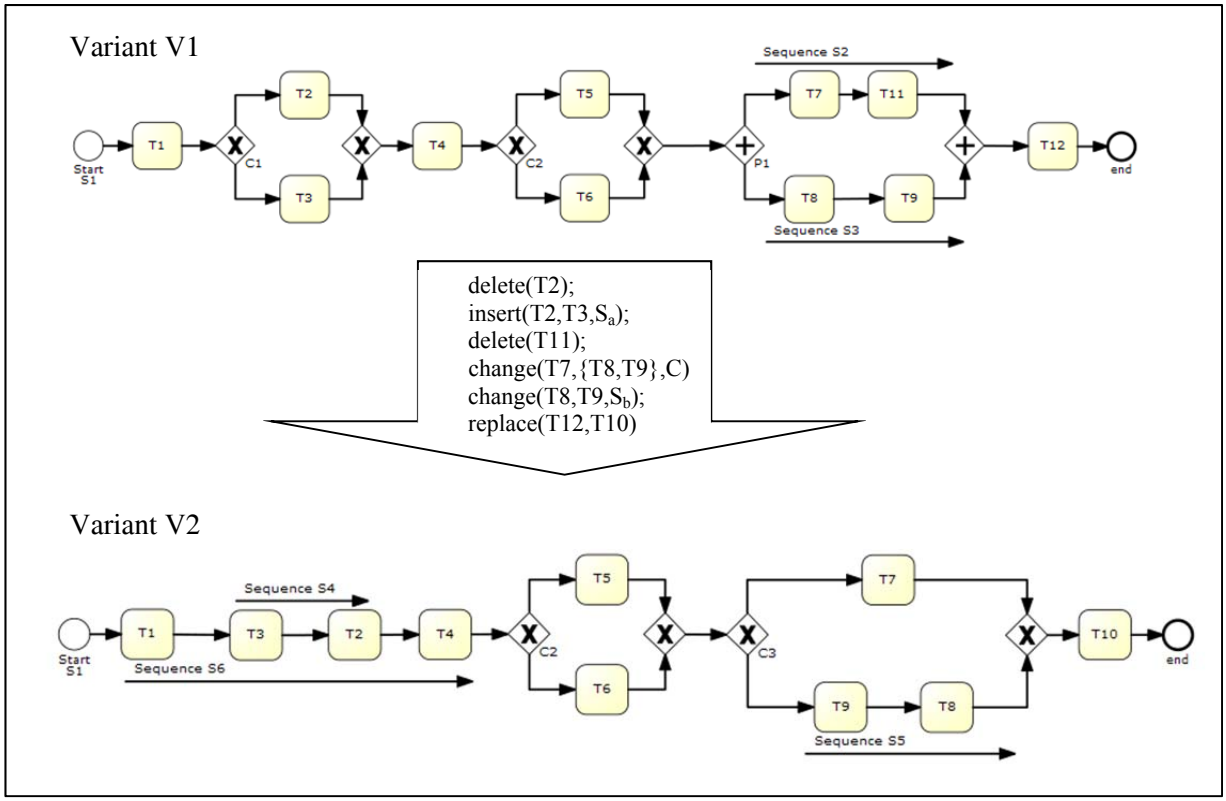


Figure 11. Variant V2 is derived from V1 by performing a series of operations

Table 6. Illustration of steps performed in transforming V1 to V2 in the example of Figure 11

Operation	POS string																
Variant V1	T1 1	T2 2	T3 2	C1 1	T4 1	T5 2	T6 2	C2 1	T7 3	T11 3	S2 2	T8 3	T9 3	S3 2	P1 1	T12 1	S1 0
Delete(T2)	T1 1	T3 2	T4 1	T5 2	T6 2	C2 1	T7 3	T11 3	S2 2	T8 3	T9 3	S3 2	P1 1	T12 1	S1 0		
insert(T2,T3,S_A)	T1 1	T3 2	T2 2	S4 1	T4 1	T5 2	T6 2	C2 1	T7 3	T11 3	S2 2	T8 3	T9 3	S3 2	P1 1	T12 1	S1 0
delete(T11)	T1 1	T3 2	T2 2	S4 1	T4 1	T5 2	T6 2	C2 1	T7 3	T8 3	T9 3	S3 2	P1 1	T12 1	S1 0		
change(T7,S3, C)	T1 1	T3 2	T2 2	S4 1	T4 1	T5 2	T6 2	C2 1	T7 3	T8 3	T9 3	S3 2	C3 1	T12 1	S1 0		
change(T8,T9,S_B)	T1 1	T3 2	T2 2	S4 1	T4 1	T5 2	T6 2	C2 1	T7 3	T9 3	T8 3	S5 2	C3 1	T12 1	S1 0		
replace(T12,T10)	T1 1	T3 2	T2 2	S4 1	T4 1	T5 2	T6 2	C2 1	T7 3	T9 3	T8 3	S5 2	C3 1	T10 1	S1 0		
Variant V2	T1 2	T3 2	T2 2	T4 2	S6 1	T5 2	T6 2	C2 1	T7 3	T9 3	T8 3	S5 2	C3 1	T10 1	S1 0		

The representation developed here will be employed in the next section in the context of managing and searching a large repository of process models.

5. Managing a Large Repository of Process Variants – Analysis and Discovery

In this section we first introduce notions of similarity among process models as a way to cluster related models. Then we discuss how a large number of process variants can be indexed using bit vector structures and accessed using SQL queries. Later we discuss metrics for evaluating the complexity of a process model and use them to compare various models.

5.1. Similarity

One notion of similarity is the number of edit operations between variants. Figure 12 shows how variants can be placed in a graph. V1 through V6 are variants derived from a base variant V by performing one or more operations on the arcs connecting them. As shown in this figure, it is also possible to derive one variant from another one (e.g. V3 from V2) by performing an operation. It is also possible to go in reverse by performing an inverse operation. For example, the two operations op1 and op2 are performed in sequence transform V to V3. The inverse operation to go from V3 to V is $(op1, op2)^{-1}$. In general,

$$(op1, op2, \ldots.opn)^{-1} = opn^{-1}, \ldots op2^{-1}, op1^{-1}.$$

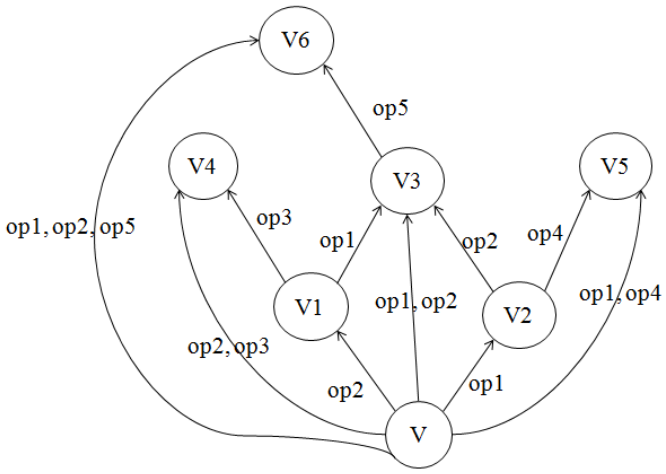


Figure 12. A graph of variants showing how one variant is derived from another by operations

However, it is difficult to relate to the notion of edit distance because it does not consider the number of tasks that are common between two processes or the structural similarity among them. Hence, we introduce two other measures called task distance and structural distance. *Task distance* is defined as follows:

$TaskDistance(i, j) =$ # of tasks that are different between process model variants *i, j*
= Hamming distance between task vectors of variants *i, j*
= # tasks (structures) present in model *i* but not in model *j* +
tasks (structures) present in model *j* but not in model *i*

The *structural distance* between variants *i, j* is based on comparing the pre-order traversal strings of the variants. We create the pre-order strings for the two variants first. Then, we remove all task entries from the strings to perform a structural comparison. For the example of Figure 9, this will result in a string like: S1(0)-S2(1)-S4(2)-S3(1)-S5(1)-P2(2). Thus, only the structural information is retained. In this structure there are five sequence nodes at levels 0, 1 and 2, and also one parallel node at level 2.

$$StructDistance(i, j) = \sum_{k=0}^{\#levels} \frac{1}{2^k} \left(distance_{level(k)}(i, j) \right)$$

where

$distance_{level(k)}(i, j)$: # of structures that differ between models *i, j* at level *k*

The idea of this formula is to compare the structural similarity at the top level, i.e. the root level, and then at successive sublevels. The $\frac{1}{2^k}$ term is intended to reduce the weight of dissimilarity or distance at successively lower levels.

These notions of similarity and distance can be used to cluster a group of variants together if their inter-process distance on various metrics is less than a cutoff value, and also to find one base variant with the minimum distance from all other variants. We do not discuss clustering algorithms here, but refer to other work (e.g. [18, 19]).

5.2. Process Variant Discovery

In this section, we discuss techniques for querying a large repository of variants using data structures and techniques. First, we maintain two bit vectors, *task bit vector* and *structure bit vector* that represent the presence of tasks and control flow structures in a variant, respectively. Thus, a task vector (0,1,1,0,1,1,0) means that this variant contains tasks T2, T3, T5 and T6. A user can query the bit vector and find the variants that meet the requirement and also the cluster in which they lie. Similarly, a structure vector of (1,0,1,0) indicates that the 'S' and 'C' control flow structures are present in the variant but 'P' and 'L' are not. These vectors are included as parts of the Variant_index table shown in Table 7.

Table 7. An example Variant_index table containing task and structure bit vectors for each variant

		Task bit vector							**Structure bit vector**			
variant	cluster	T1	T2	T3	T4	T5	T6	T7	S	P	C	L
V1	C1	1	0	0	1	1	0	0	1	0	1	0
V2	C1	1	1	0	1	0	1	0	1	1	1	0
V3	C2	1	1	1	1	1	1	1	1	0	1	0
V4	C2	1	1	1	0	1	1	1	0	0	1	0
V5	C1	1	1	0	1	0	1	1	1	1	0	0
V6	C2	1	0	1	1	1	1	1	1	0	1	0

The Variant_index table can be queried using SQL. Some simple example queries are as follows:

Q1: **Select** variant from Variant_index **where** T1 =1 and T2 = 1 **and** T7 = 0
Q2: **Select** variant from Variant_index **where** T1 = T2 **and** T3 = 1 **and not**(T6 = T7)
Q3: **Select** variant from Variant_index **where** T1+ T2+T3 ≤ 2 **or** T6 + T7 = 1
Q4: **Select** variant from Variant_index **where** Task_distance < 5 **and** Struct_distance < 3
Q5: **Select** count(*) from Variant_index **where** T1 = 1
Q6: **Select** count(*)from Variant_index **where not**(T1 = T2)

In this way a large variety of queries can be designed using the standard Boolean operations. These queries can be processed within a Relational Database Management System (RDBMS). More advanced queries can relate to the control flow structures present in a variant. A user may wish to find all variants where tasks Ti and Tj appear in parallel, sequence, etc. Such a query can be expressed as:

Q7: **Select** Cluster, Model **where** T1 =1 **and** T2 = 1 **and** T7 = 0 and Parallel(T1, T2)

The algorithm to process such a query is shown in Figure 13.

Algorithm Relationship_Query (Variant_index, *query_clause*, *POS[]*)
1. Rewrite the query clause so that a relationship X(Ti,Tj) is replaced by X=1
2. Find all variants where the rewritten query clause is satisfied.
3. Then transform the set of resulting variants into **in-order** traversal strings.
4. Extract the substring that lies between T1 and T2 in each string.
5. Find the highest level control structure in this substring.
6. If it is P then this variant is included in the result of the query; otherwise not.

Figure 13. Algorithm to determine the relationship between two nodes by scanning POS string

The algorithm rewrites the query clause to check the structure vector for even a single occurrence of the desired structure. In query Q7, Parallel(T1, T2) would be replaced by P=1, and the query condition is rewritten as: "T1 =1 and T2 = 1 and T7 = 0 and P=1." Then it searches the bit vector for matching variants. Then the POS string for each variant is scanned to determine the relationship between a pair of nodes and the ones that do not satisfy the query are filtered out. If the query contains multiple relationship conditions then the scan is repeated for the remaining set of variants. The order of screening for relationships can be arranged so that the variants are first screened for the relationship that is less likely. As an example of screening, consider the in-order traversal (which is a variant of the post-order traversal) of the process of Figure 9:

In-order:T1(3)- S4(2)-T2(3)-S2(1)-T3(2)-S1(0)-T4(2)-S5(1)-T5(2)-S3(1)-T6(3)-P2(2)-T7(3)-T8(2)

Now, to find the relationship, say, between T6 and T7, we extract the substring that includes both T6 and T7, i.e. T6(3)-P2(2)-T7(3). The highest level of control structure in this string is P2. So the relationship between T6 and T7 is of type parallel. Similarly, for task T4 and T8, we extract the substring that includes both T4 and T8 as: T4(2)-S5(1)-T5(2)-S3(1)-T6(3)-P2(2)-T7(3)-T8(2). The highest control structure in this is S3 at level 1. This means that the relationship between T4 and T8 is of type sequence.

5.3. Querying a Rulebase

A second approach for finding variants is by querying the rulebase on case data. For instance, a query in the insurance handling repository may state:

Q8: **Select** * from Rules **where** loss > 500K **and** status = "urgent"

This will find the process variants in the collection that handle cases with loss larger than $500K and in *urgent* status. Again the Boolean operators can be used to connect atomic predicates and create more powerful search conditions. A more complex query can be specified as:

Q9: **Select** * from Rules **where** loss > 300K **and** loss < 500K **and** (NOT (status = "urgent"))

or (loss > 500K AND status = "normal")

It is also possible to write queries pertaining to the consequent of a rule, i.e. the action tasks required to be performed as a result of a rule. This query returns the action tasks that require the role manager. These types of queries can be implemented by using standard database techniques and by building indexes on important attributes that appear frequently in rules, such as loss, status and role in our running example. Querying and searching is an important aspect of managing a large number of variants. Both the representation of data objects, as well as the nature of indexing structures, plays an important role in determining the effectiveness of search. Our representation scheme facilitates string searches to look for patterns and variants. Any given pattern or variant can itself be described as a post-order traversal string. Then, it is possible to search for this string in a file of variants for exact match using any text editor. It is also possible to search for partial matches using a partial match score of similarity. Moreover, the bit vector index allows us to narrow down the number of variants on which string search needs to be carried out. The operations on such index tables can be performed very fast. In addition, exact string searches on very large text files (or, in our case, on a large number of rows of strings) can be also carried out in sub-second times. Therefore, the scheme proposed here is promising.

5.4. Evaluation of Process Complexity Reduction

Our study can facilitate the design and management of large collections of process models. In this section, we evaluate our approach by measuring the complexity of process variants versus a single process model using a number of standard complexity metrics. Gaining insights from software engineering, cognitive science, and graph theory, Cardoso et al. [20] proposed a variety of process complexity metrics. These metrics includes *number of activities, number of activity and control nodes, McCabe's cyclomatic complexity*, and *control-flow complexity (CFC)*. CFC concerns the process structure or control flow patterns. It has significant impact on process design as shown by empirical evidence [20]. The CFC of a process P is defined as:

$$CFC(P) = \sum_{c \in XOR\_split} CFC_{XOR}(c) + \sum_{c \in OR\_split} CFC_{OR}(c) + \sum_{c \in AND\_split} CFC_{AND}(c)$$

where

C is a control node of type XOR-split, OR-split, AND-split
$CFC_{XOR}(c) = fan\text{-}out\ (c)$
$CFC_{OR}(c) = 2^{fan\text{-}out\ (c)} - 1$
$CFC_{AND}(c) = 1$

Another set of metrics proposed by Mendling and Neumann [21] aims to identify errors in process design. These metrics capture the degree of sequentiality, cyclicality, and parallelism in a process model and hence reflect its complexity that in turn relates to the likelihood of errors in a process model [21]. For example, an increase in sequentiality would imply a decrease in the probability of errors in a model, while an increase in cyclicality would imply an increase. The complexity of process models is related to the quality of process models, as elaborated in [22].

In order to comprehensively evaluate the complexity of two variants, V1 and V2 in Figure 3, against the integrated process model in Appendix 1, we use a combination of metrics from the abovementioned studies which have been empirically validated.

Table 8. Process complexity metrics and the evaluation results

Complexity Metrics		**Single model**	**Process Variant**	
Perspective	**Metrics description**	**P**	**V1**	**V2**
Process size (Cardoso, 2006)	Number of activity nodes	26	8	8
	Number of activity and control nodes	43	12	10
	McCabe's cyclomatic complexity	13	2	1
Control-flow perspective (Cardoso, 2006; Mendling and Neumann, 2006)	Number of XOR-split nodes	13	0	0
	Number of OR-split nodes	0	0	0
	Number of AND-split nodes	3	1	0
	Control-flow complexity (CFC)	16	1	0
	Sequentiality	2/9	1/2	1
	Cyclicity	2/27	0	0
	Parallelism	3	1	0

Table 8 summarizes the complexity metrics used in this study and our evaluation results. From this table, we can see that even for a process with ten tasks, the complexity of a single model that integrates six rules R1-R6 (R7, R8, and R9 are excluded since they concern the data flow) is much greater than that of the variants generated by our approach. Although our variants are also maintained as process models in the repository, they are tightly associated to the template through rules. Thus, they will be automatically adapted once any rules are modified in response to policy changes.

6. Prototype Implementation

In this section, we introduce the implementation architecture and our proof-of-concept prototype. We show the feasibility and applicability of our approach by walking through the running example of insurance claim processing in our prototype system.

6.1. System Architecture

A high level architecture for our approach is shown in Figure 14. A process designer can create, modify or delete process templates and rules using an editor. As described above, a process template, relevant rules and variants are organized in a cluster, which is a process family that refers to an application domain (e.g., insurance claim handling, patient workflow, etc.). Generally, each cluster only contains one template represented in BPEL but may contain hundreds or thousands of variants. The process template is configurable for generating more variants by using a number of rules. The editor checks the template for correctness, and the rules for consistency. We implemented a prototype in Java and used the Drools Rule Language (DRL) to represent our rules. The open source rule engine Drools-expert [23] is used to process rules and resolve conflicts.

The process designer can easily configure a process variant using the *variant designer*. First, he needs to select a process template and then feed case data as a variant scenario. Then the *rule engine* produces a valid set of change operations according to the received case data. Specifically, the *rule processing module* is responsible for firing rules and coordinating with the other two components. It makes sure the resulting change operations are valid and correct before they are passed on to the variant configuration algorithm. In the case where two rules conflict, the *conflict resolution component* will produce a warning and ask the designer either to modify the rule or assign priorities to them. The *validity checking component* ensures the resulting change operations can be applied to the process template in terms of structural correctness using the verification matrix in Table 3. For example, the editor will produce an error or a warning to the process designer if the action delete (task t1) is triggered but t1 does not exist in the process template. Hence, valid process templates and rules are maintained respectively in their own repositories. As the business policy changes over time, the process designer can change rules associated with a specific process while keeping the process template the same.

The *variant configuration algorithm* uses these operations to modify the process template, create a variant schema, and store it in the repository. This is enabled by the transformation between a process template in BPEL and its corresponding process tree. For each process tree, we maintain a (task vector, level vector) pair. Further, the interchangeable representations of the tree are maintained, as well as post-order, in-order and pre-order structures. We use post-order traversal for making changes to the process template. After a variant is generated, the rule editor can also check for *data flow consistencies* to ensure that a task will receive all its input data from the output of previous tasks. Otherwise, an error is flagged. Besides configuring a variant from scratch, we also consider *process refactoring* [24] from an existing set of process models. Process refactoring includes extracting process templates as well as rules from either a large single model or multiple redundant models. This can be helpful for industry to manage the transition

from the existing process management methodology to our proposed methodology. We plan to implement this part as a plug-in into our existing prototype. It should be noted that a domain expert should be assigned to validate the compliance (e.g., to certain norms) of the derived process variants before they can be deployed.

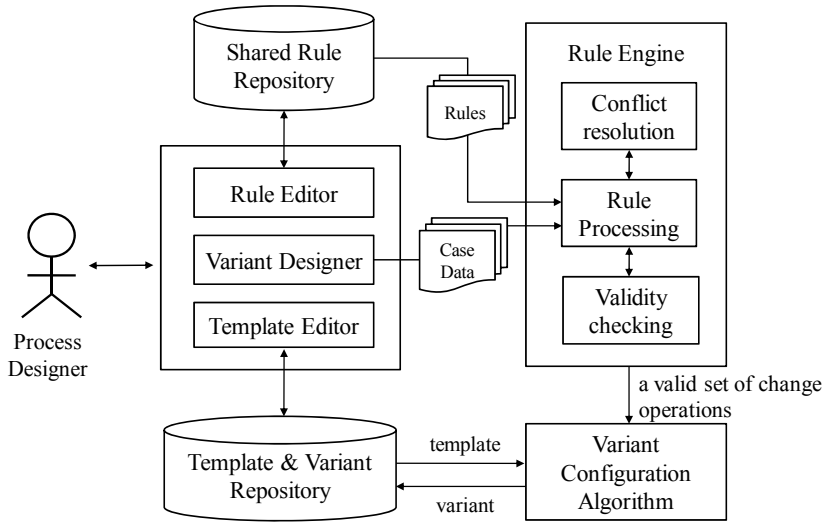


Figure 14. Architecture of process variant configuration

After a variant is validated, it is ready to be enacted and executed by the workflow engine. However, the user needs to select the appropriate variant that fits the application scenario. For example, to handle a car damage claim with a loss of $10k with "normal" status, our system should automatically find the variant that fits the scenario and present it to the user. Figure 15 presents our architecture for variant search, discovery and instantiation. We implement two types of query processing for variant discovery: Boolean query and scenario-based query. We run a query against the Variant_index table that maintains presence of tasks and control-flow structures in a variant. This is helpful when a user wants to find a number of variants that share the same tasks or similar structure. In contrast, a scenario-based query has more semantics and is goal oriented. It compares the case data with the rule conditions, and finds the matching rules that lead to their process variants. After a process instance is completed, the frequency of its use is automatically updated in a log kept in the database. These statistics can be helpful in various ways, e.g. variants with low usage frequency can be discarded.

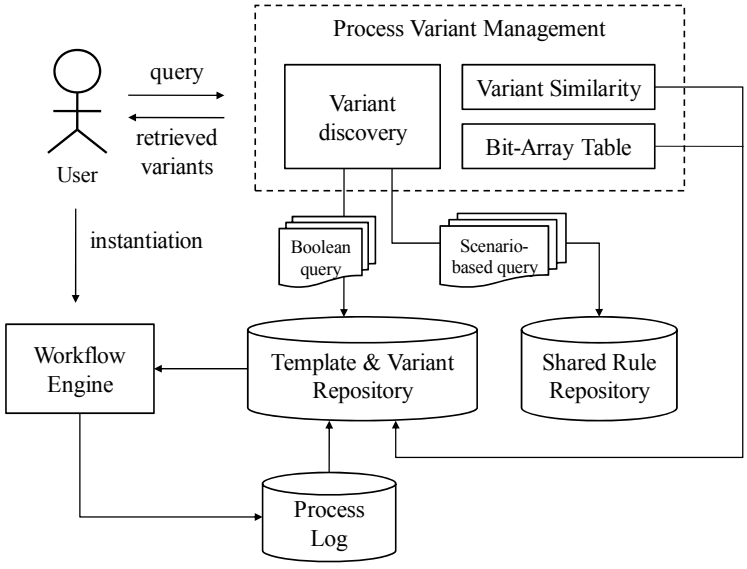


Figure 15. Architecture for variant search, discovery, and instantiation

6.2. A Use Case for Handling Insurance Claims

In this section, we will walk through the running example of insurance claim handling in our prototype. Figure 16 shows the user interface of the process variant manager. Three process clusters are deployed for an insurance company, say ABC Corp. Clusters represent sets of relevant templates and variants in different application domains. For example, C1 is used to start a new insurance quote, C2 to handle insurance claims, and C3 to process customer complaints. Say, John is a process designer who wishes to implement the motivating scenarios. When he clicks on cluster C2, the right panel shows the *similarity values* between variants and their corresponding template. The columns in this comparison table include *task distance, structure distance, usage frequency* and *application scenario*. Depending on the template design preferences a template can be considered as the most frequently used process model and it can be instantiated. It is also possible that a template is a minimum set of required tasks across all variants, and is not be applicable to any application scenario without some modification. Figure 16 shows that the process template V has been instantiated five times.

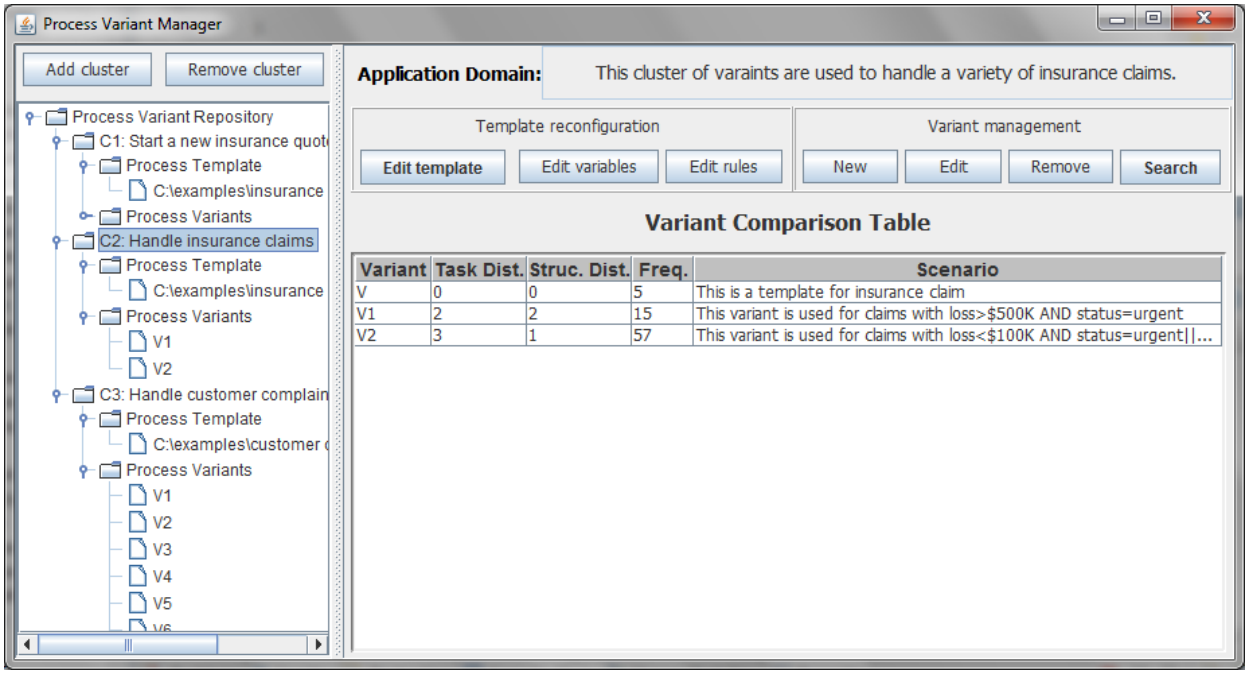


Figure 16. User interface of process variant manager

Figure 17 shows the user interface for creating a new variant (i.e., by clicking on the "new" button in the variant configuration group in Figure 16). The process designer can examine the process template by browsing the BPEL structure in the left panel, and the process tree in the right panel. John can also learn about the current rules by clicking the RuleSet tab. Moreover, he can directly input the change operations and the application scenario based on heuristics, or he can also generate a variant by feeding case data, as shown in Figure 18 (a). The resulting process variant is derived from the modified process tree shown in Figure 18 (b), and then added to the process repository. After a specific variant is generated, John should send it to a senior employee who is familiar with insurance business logic in ABC Corp. for evaluation.

Over time, the business policies in ABC Corp. may evolve. Upon receiving change requirements, John can revise the rules so that changes propagate to all the insurance processes, instead of manually revising them one at a time in the traditional way. After variables or rules are revised, John is responsible for running the variant configuration to ensure the current process is compliant with up-to-date rules. Existing running instances may have to be restarted, continued without change, or migrated to new variants from the current time.

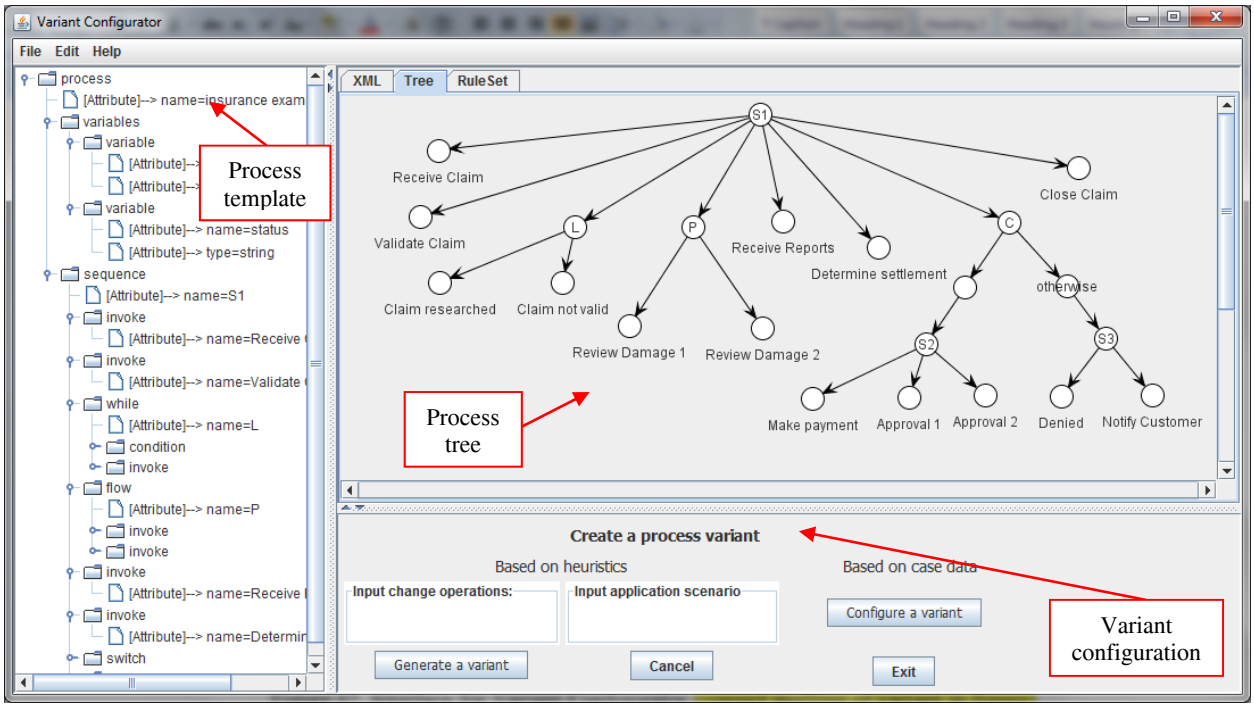
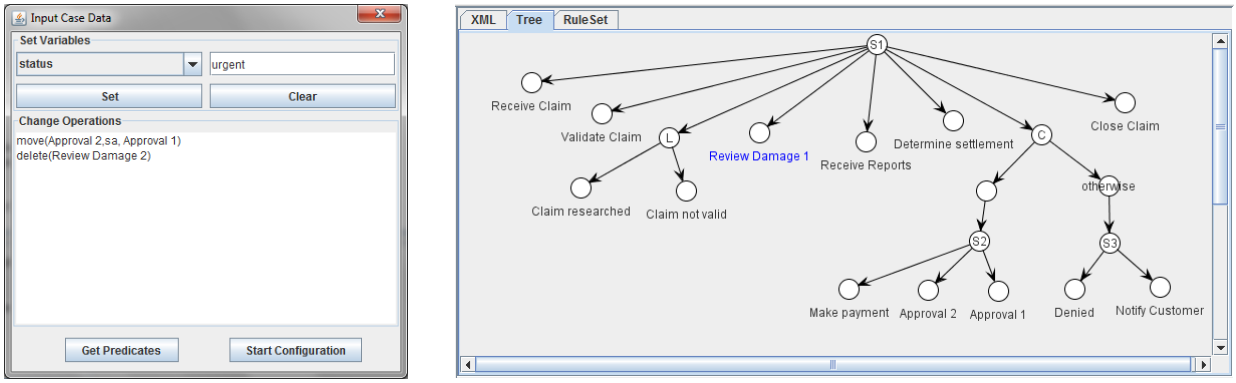


Figure 17. Interface for Variant Configurator



(a) Input case data (b) Revised process tree

Figure 18. Process variant configuration

6.3. Discussion and Limitations

This use case shows how to use our prototype to design and manage variants based on the algorithms we proposed. We assume the template and rules are designed from scratch. For companies that already have a large repository of redundant or complex process models, we need an efficient and automated technique for refactoring process templates and rules from existing models. For a large and complex process model, we need to identify the repetitive elements and merge them into a concise structure with the rules for

dealing with variations such as, for example, multiple branches. The methodology is similar for handling multiple redundant models but it is applied to different process models. Thus, semantic inconsistency of task names or labels should be resolved.

As business processes and policies evolve, they affect process models. One advantage of our approach is that it facilitates the management of process model evolution. Since process variants are derived from a template, any change that is made to the template (e.g. insert or remove a task) will automatically propagate to its variants. Handling change to business policy is also made simpler. As noted above, it is time and effort consuming if process variants are designed as different process models. If any policy changes, each model related to that policy must be changed as well. However, in our approach, we only need to modify the rules that embody that policy. It is still necessary to check consistency of the rules after any change is made. Nonetheless, it is still easier than checking every process variant, applying the new policy to it and then verifying the correctness of each revised process. It also offers more flexibility.

We recognize our approach has several limitations. First, we only restrict ourselves to structured processes. Some business processes are unstructured. In such situations they have to be mapped into equivalent structured processes, if possible. New techniques for managing unstructured processes are also required. Second, it is necessary for users to specify the semantics of the applied rules. In Section 3.3, we discussed possible semantics for conflict resolution when two or more rules may apply to a given case data. In Section 3.4, we developed a matrix to disallow invalid combinations of operations on a template. However, the checking of the rule semantics still has to be done manually by the user to ensure that a rule does not produce unintended consequences. Third, additional cost is involved in managing the rules and ensuring their consistency. If the rule semantics is incorrect it can lead to variants that are wrong. In this respect it is important to realize that the power of the configuration operations is a double-edged sword. On the one hand they give considerable flexibility, but on the other they enable creation of variants that may be incorrect. Consequently, it may be useful to develop additional controls in the form of meta-rules to somewhat restrain the flexibility of the approach. Finally, the goal of a template is to capture the "essence" of a process in such a way that other variants can be derived easily from it. However, it would be very helpful to have a precise definition of what is meant by a template. One possible definition could be based on the notion of similarity as discussed in Section 5.1, but the difficulty lies in that the possible variants that may arise from a template, along with their frequencies of occurrence, is hard to estimate.

7. Related Work

7.1. Process Instance Adaptation at Runtime

Techniques for supporting dynamic changes at the process instance level are discussed in [8, 9] and elsewhere. The focus of this work is to allow operations like task insertion, deletion, etc. to be performed on running workflows in response to exceptions [25-27]. There have also been other approaches on designing flexible workflow models: based on deadline based escalation [28], satisfaction of constraints [29] or restrictions [30], availability of resources [31] and on graphs [32]. Yet another approach for designing workflows is centered on entities [33]. While all these methods try to reduce rigidity of a strict control flow approach, they lack the flexibility of rules. Müller et al. [8] proposed a rule-based approach for dynamic modification in a medical domain with the focus on handling of exceptions. Rules can also be used to ensure that business processes comply with internal and external regulations. These rules could be specified in a first-order language [34] or in deontic logic [35].

There is considerable amount of related work on runtime process instance adaptation, especially in the context of exception handling. The focus there is on modifying a running process when exceptions occur due to failed tasks, erroneous information, etc. However, this body of work is complementary to our work since it focuses on flexibility at run time, as opposed to at design time as in our work.

7.2. Configurable Process Models and Aspect-oriented Process Modeling

Recent years have witnessed a growing interest in enabling flexibility at the process model level. Most process design techniques lead to *rigid* processes where policy is "hard-coded" into the process schema thus reducing flexibility. As a result, a process oriented software system may contain a family of similar but customizable processes [36]. A number of approaches are proposed to tackle this issue. For example, Schnieders and Puhlmann [36] proposed an approach to manage variability implementation using Java variability mechanisms and code generators. A multi-layered approach for configuring process variants from the base layer is presented in [37]. Hallerbach et al. [6, 38] proposed the Provop approach that allows a user to design a base process (i.e., a process template) with various options. For example, a user can adopt the most frequently used process models as a template or extract the minimum common structure as a template, etc. Another research stream models process variations as a workflow version control problem, and many studies have been conducted along this line [39-41].

Reference process models are proposed to handle the large number of processes with variations [42]. At design time, variation points are assigned to a reference process model. Then domain experts have to configure the variation points according to their own needs manually. In reality, this approach is error prone because people who are familiar with business policies may have no knowledge about process

modeling. Thus, a questionnaire-based guidance component [43] is used to lead an end user to configure process variants based on formal conceptualization of domain knowledge. The correctness-preserving approach to handle syntactic and semantic correctness during configuration is proposed and discussed by van der Aalst et al. [44].

Inspired by Aspect-Oriented Programming (AOP) in software engineering, Charfi and Mezini [13] proposed an aspect-oriented workflow language to enable a concern-based decomposition of process specifications and process models. According to the principle of separation of concerns, the process logic is encapsulated in a process module whereas crosscutting concerns are encapsulated in aspect modules. Crosscutting concerns are called "aspects," and these are shared among different process models, such as compliance, accounting, billing, authorization, etc. They also proposed AO4BPEL [14] and AO4BPMN [15] that apply the idea of aspect-oriented process modeling in BPEL and BPMN languages respectively. AO4BPEL [14] uses AOP to support a modular and dynamic strategy for web service composition. It defines an aspect-oriented extension to BPEL where aspects and processes are expressed in XML. Further, it defines a pointcut language to capture the join points that span several processes. AO4BPMN [15], on the other hand, adds aspect-oriented extensions to BPMN in a similar way.

7.3. Business Processes and Rules

Business rules have been used to improve the expressiveness of existing process modeling languages. Goedertier and Vanthienen proposed to use process and rule set meta models to represent the semantics of case data and activity flows in business process models [45]. Later, they proposed declarative process modeling with business vocabularies and business rules in [7]. A preliminary proposal for process and rule integration is given in [12]. Business rules can be categorized into integrity rules, deviation rules, reaction rules and Deontic assignments. In [11], a service oriented approach has been proposed to integrate business rules into BPEL. This approach is independent of any rule language and it allows the separation of business logic from process logic. Other related works with similar objectives pertain to configurable models [42] and aggregate models [46] although they are not based on rules directly. Direct use of rules to model the variable parts of process flow and facilitate dynamic pattern composition in complex business processes is discussed in [10]. In aspect-oriented process modeling, "aspect" can be considered as a form of business rules that are integrated with a BPEL orchestration engine. Integration of the process and rule paradigms into a product is being made in the Drools Flow project [47]. Their goals are similar to ours, but they provide a rule modeling construct and tightly integrate rules in the process model. Although they are quite strong on rule modeling and validation, their process models are inflexible since rules are embedded within the process model.

Most rule-based approaches aim to improve the expressiveness and flexibility of existing process modeling languages by using rules to handle the knowledge-intensive parts of a business process. In contrast, rules used in our approach can adapt various process patterns instead of certain decision points. One advantage of our approach is that it can be integrated into existing process modeling paradigms, because the change operations we developed are independent of any workflow modeling languages. Upon configuration, our process models can be expressed in any existing language, such as BPMN and BPEL, and executed in an existing workflow engine. Although we use BPEL to model generic process templates, they can also be written in any standard language. Further, this study not only uses rules for configuring process variants but also facilitates the search and retrieval of desired variants in a large repository by querying a rulebase.

7.4. Variant Management and Search

As discussed above, there is considerable amount of research on designing flexible processes and configurable process models. Work has also been done on linking rules with processes. But surprisingly, there is only a limited amount of research in actually managing a large number of variants. An initial proposal for querying business workflows in XML was described in [48]. Here various query patterns for accessing workflows were written in the XML Query language. Another stream of research that has inspired our work is [6, 38, 49]. The main focus in these studies is on configuration of flexible workflows with only limited coverage of querying issues. The perspective taken in these works is to find variants that are "similar" to a given variant or to a given query pattern. This gives rise to the need for various metrics of similarity. For example, Dijkman et al. [19] present three similarity metrics used to answer queries on a large process repository, including node matching, and structural and behavioral similarity.

From our point of view, management of process variants encompasses two important aspects: (1) storing a large number of process variants (say, 100K or 1 million) in a repository; and (2) providing convenient and fast access paths such as indexing structures to the variants based on a query language that meets the common needs of users.

We also find that no query language standard for querying workflows has emerged. In addition, work is lacking on development of indexing structures for fast access to workflow repositories. A notable exception is the recent work of Kunze and Weske [50] where the authors proposed an indexing structure called metric trees for accessing process graphs based on similarity. Yan et al. [51] presented a technique to search model features. Their experiments showed it helps to retrieve similar models at least 3.5 times faster without impacting the quality of the results, and 5.5 times faster if a quality reduction of 1% is acceptable. There is a critical need for more work to develop efficient ways to query workflows.

8. Conclusions

This paper described a novel proposal for designing flexible business processes based on combining process templates with business rules. We showed how the template- and rule-based approach allows separation of basic process flow from business policy elements in the design of a process and also integrates resource and data needs of a process tightly. We also developed a novel scheme for storing process variants as strings based on a post-order traversal of a process tree and attaching level numbers to each node of the tree. We showed that such a representation lends itself well to manipulation and also for searching a repository of variants. Moreover, indexing structures were developed for faster access to process models based on user requirements expressed through SQL queries. Finally, we showed that it is also possible to query the rulebase to find variants that match the criteria specified in a query.

A proof-of-concept prototype has been partially implemented to test and evaluate this methodology based on the proposed architecture. However, more experiments are required to evaluate the querying and searching technique. A further challenge lies in creating an interface that allows users to describe the rules in an easy way and making the details of the language completely transparent to them. We intend to explore various solutions for this including the use of an English-like rule language, such as SBVR [52] , and providing a GUI interface to increase ease of use and prevent typographical errors. More effort will also be devoted to enhancing the richness of the process description language with events, and also on the semantics for rule conflict resolution. Lastly, adding ontologies to the architecture would increase the expressive power of the framework as well.

Acknowledgment

This work was supported in part by the Smeal College of Business at Penn State.

References

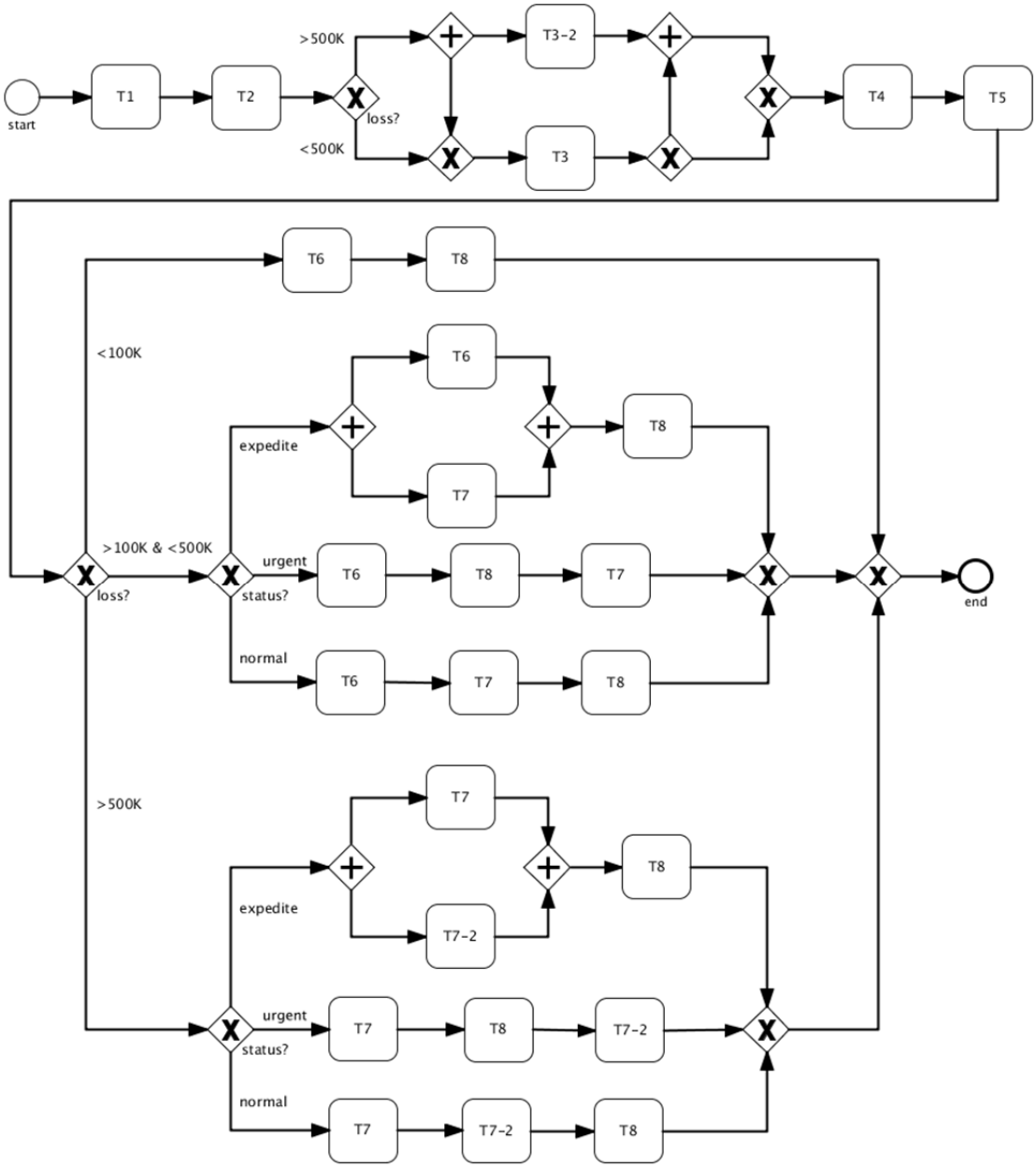
[1] W.M.P. van der Aalst, The Application of Petri Nets to Workflow Management, The Journal of Circuits, Systems and Computers, 8 (1) (1998) 21-66.
[2] W.M.P. van Der Aalst, A.H.M. Ter Hofstede, B. Kiepuszewski, A.P. Barros, Workflow Patterns, Distributed and Parallel Databases, 14 (3) (2003) 5-51.
[3] M. Dumas, W. Van Der Aalst, A. Ter Hofstede, Process-aware information systems, Wiley Online Library, 2005.
[4] B. Kiepuszewski, A. ter Hofstede, C. Bussler, On structured workflow modelling, in: B. Wangler, L. Bergman (Eds.), Advanced Information Systems Engineering, Springer, 2000, pp. 431-445.
[5] OMG, Business Process Modeling Notation (BPMN) Version 1.0. OMG Final Adopted Specification, in, Object Management Group, 2006.
[6] A. Hallerbach, T. Bauer, M. Reichert, Capturing variability in business process models: the Provop approach, Journal of Software Maintenance and Evolution: Research and Practice, 22 (6-7) (2010) 519–546.

[7] S. Goedertier, J. Vanthienen, Declarative process modeling with business vocabulary and business rules, in: Z. Tari, P. Herrero (Eds.), OTM 2007 Workshops, Springer, 2007, pp. 603-612.
[8] R. Müller, U. Greiner, E. Rahm, Agentwork: a workflow system supporting rule-based workflow adaptation, Data & Knowledge Engineering, 51 (2) (2004) 223–256.
[9] M. Reichert, P. Dadam, ADEPT flex—supporting dynamic changes of workflows without losing control, Journal of Intelligent Information Systems, 10 (2) (1998) 93-129.
[10] T. van Eijndhoven, M.E. Iacob, M.L. Ponisio, Achieving business process flexibility with business rules, in: 12th International IEEE Enterprise Distributed Object Computing Conference, Munich, Germany, 2008, pp. 95-104.
[11] F. Rosenberg, S. Dustdar, Business rules integration in BPEL-a service-oriented approach, in, Seventh IEEE International Conference on E-Commerce Technology (CEC'05) 2005, pp. 476-479.
[12] H. Lienhard, U.M. Künzi, Workflow and Business Rules: a Common Approach, Workflow Handbook, (2005) 129-140.
[13] A. Charfi, M. Mezini, Aspect-oriented workflow languages, On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, (2006) 183-200.
[14] A. Charfi, M. Mezini, Aspect-oriented web service composition with AO4BPEL, in: European Conference On Web Services (ECOWS), Erfurt, Germany, 2004, pp. 168–182.
[15] A. Charfi, H. Müller, M. Mezini, Aspect-oriented business process modeling with AO4BPMN, Modelling Foundations and Applications, (2010) 48-61.
[16] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, others, Business process execution language for web services, version 1.1, Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation, (2003).
[17] A. Kumar, W. Yao, Process Materialization Using Templates and Rules to Design Flexible Process Models, in: G. Governatori, J. Hall, A. Paschke (Eds.), International Symposium on Rule Interchange and Applications (RuleML'09), Springer, Las Vegas, Nevada, USA, 2009, pp. 122-136.
[18] C. Li, M. Reichert, A. Wombacher, Mining business process variants: Challenges, scenarios, algorithms, Data & Knowledge Engineering 70 (5) (2011) 409-434
[19] R. Dijkman, M. Dumas, B.v. Dongen, R.K. Mendling, Similarity of business process models: Metrics and evaluation, Data & Knowledge Engineering, 36 (2) (2011) 498-516.
[20] J. Cardoso, J. Mendling, G. Neumann, H. Reijers, A discourse on complexity of process models, in: Business Process Management Workshops, 2006, pp. 117-128.
[21] J. Mendling, G. Neumann, Error metrics for business process models, in: 19th International Conference on Advanced Information Systems Engineering (CAISE 2007), Trondheim, Norway, 2007.
[22] I. Vanderfeesten, J. Cardoso, J. Mendling, H.A. Reijers, W. van der Aalst, Quality metrics for business process models, BPM and Workflow handbook, (2007) 179-190.
[23] JBoss Community, Drools Expert, in, http://www.jboss.org/drools/drools-expert, 2011.
[24] B. Weber, M. Reichert, Refactoring Process Models in Large Process Repositories, in: Z. Bellahsène, M. Léonard (Eds.), Advanced Information Systems Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 124-139.
[25] D.K.W. Chiu, Q. Li, K. Karlapalem, Web interface-driven cooperative exception handling in ADOME workflow management system, Web Information Systems Engineering, 26 (2) (2001) 93-120.
[26] F. Curbera, R. Khalaf, F. Leymann, S. Weerawarana, Exception handling in the BPEL4WS language, in: W. M. P. van der Aalst et al. (Ed.), Business Process Management (BPM'03), 2003, pp. 276 -290.
[27] S.Y. Hwang, J. Tang, Consulting past exceptions to facilitate workflow exception handling, Decision Support Systems, 37 (1) (2004) 49–69.
[28] W.M.P. van der Aalst, M. Rosemann, M. Dumas, Deadline-based escalation in process-aware information systems, Decision Support Systems, 43 (2) (2007) 492-511.
[29] P. Mangan, S. Sadiq, On building workflow models for flexible processes, Australian Computer Science Communications, 24 (2) (2002) 103-109.

[30] J.J. Halliday, S.K. Shrivastava, S.M. Wheater, Flexible workflow management in the OPENflow system, in Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing (EDOC'01), 2001, pp. 82-92.
[31] A. Kumar, J. Wang, A framework for designing resource driven workflow systems, in: J.v. Brocke, M. Rosemann (Eds.), Handbook on Business Process Management 1, Springer, 2009, pp. 419-440.
[32] M. Weske, Flexible modeling and execution of workflow activities, in, Proceedings of the Thirty-First Hawaii International Conference on System Sciences, Kohala Coast, HI , USA 1998, pp. 713-722.
[33] K. Bhattacharya, C. Gerede, R. Hull, R. Liu, J. Su, Towards formal analysis of artifact-centric business process models, in: G. Alonso, P. Dadam, M. Rosemann (Eds.), Business Process Management (BPM'07), Brisbane, Australia, 2007, pp. 288-304.
[34] A. Kumar, R. Liu, A rule-based framework using role patterns for business process compliance, in: N. Bassiliades, G. Governatori, A. Paschke (Eds.), The International RuleML Symposium (RuleML'08), Orlando, FL, USA, 2008, pp. 58-72.
[35] S. Goedertier, J. Vanthienen, Designing compliant business processes with obligations and permissions, in: J. Eder, S. Dustdar (Eds.), Business Process Management Workshops, Vienna, Austria, 2006, pp. 5-14.
[36] A. Schnieders, F. Puhlmann, Variability mechanisms in e-business process families, in, 9th International Conference on Business Information Systems (BIS 2006), 2006, pp. 583-601.
[37] M. Nakamura, T. Kushida, A. Bhamidipaty, M. Chetlur, A Multi-layered Architecture for Process Variation Management, in, World Conference on Services - II (SERVICES-2 '09), 2009, pp. 71-78.
[38] A. Hallerbach, T. Bauer, M. Reichert, Configuration and management of process variants, in: J.v. Brocke, M. Rosemann (Eds.), Handbook on Business Process Management 1, 2010, pp. 237-255.
[39] H. Bae, E. Cho, J. Bae, A version management of business process models in bpms, in: K.C.-C. Chang, W. Wang, L. Chen, C.A. Ellis, C.-H. Hsu, A.C. Tsoi, H. Wang (Eds.), Advances in Web and Network Technologies, and Information Management, 2007, pp. 534-539.
[40] X. Zhao, C. Liu, Version Management in the Business Process Change Context in: G. Alonso, P. Dadam, M. Rosemann (Eds.), 5th International Conference on Business Process Management (BPM'07), Brisbane, Australia, 2007, pp. 198-213.
[41] O. Thomas, Design and implementation of a version management system for reference modeling, Journal of Software, 3 (1) (2008) 49.
[42] M. Rosemann, W.M.P. van der Aalst, A configurable reference modelling language, Information Systems, 32 (1) (2007) 1-23.
[43] M. La Rosa, M. Dumas, Configurable Process Models: How To Adopt Standard Practices In Your How Way?, BPTrends Newsletter, (2008).
[44] W.M.P. Aalst, M. Dumas, F. Gottschalk, A.H.M. Hofstede, M.L. Rosa, J. Mendling, Preserving correctness during business process model configuration, Formal Aspects of Computing, 22 (3) (2009) 459-482.
[45] S. Goedertier, J. Vanthienen, Rule-based business process modeling and execution, in, Proceedings of the IEEE EDOC Workshop on Vocabularies Ontologies and Rules for The Enterprise, 2005.
[46] H.A. Reijers, R.S. Mans, R.A. van der Toorn, Improved model management with aggregated business process models, Data & Knowledge Engineering, 68 (2) (2009) 221–243.
[47] JBoss Community, Drools Flow, in, http://www.jboss.org/drools/drools-flow.html, 2011.
[48] V. Christophides, R. Hull, A. Kumar, Querying and Splicing of XML Workflows, in: C. Batini, F. Giunchiglia, P. Giorgini, M. Mecella (Eds.), 9th International Conference on Cooperative Information Systems (CoopIS'01), Trento, Italy, 2001, pp. 386 - 403.
[49] R. Lu, S. Sadiq, G. Governatori, On managing business processes variants, Data & Knowledge Engineering, 68 (7) (2009) 642-664.
[50] M. Kunze, M. Weske, Metric Trees for Efficient Similarity Search in Process Model Repositories, in, Proceedings of the 1st International Workshop "Process in the Large", Hoboken, New Jersey, USA, 2010.

[51] Z. Yan, R. Dijkman, P. Grefen, Fast Business Process Similarity Search with Feature-Based Similarity Estimation, in: R. Meersman, T. Dillon, P. Herrero (Eds.), On the Move to Meaningful Internet Systems (OTM 2010), Hersonissos, Crete, Greece, 2010, pp. 60-77.
[52] S. Goedertier, C. Mues, J. Vanthienen, Specifying process-aware access control rules in SBVR, in: A. Paschke, Y. Biletskiy (Eds.), The International RuleML Symposium (RuleML'07), 2007, pp. 39-52.

Appendix 1: A single model that integrates Rules R1-R6



Note: some tasks have to repeated because there is no easy way to model all scenarios using control nodes

Notation:

T1: Receive claim	T5: Determine settlement
T2: Validate claim	T6: Approval 1 (by manager)
T3: Review damage 1	T7: Approval 2 (by senior manager)
T3-2: Review damage 2	T7-2: Approval 4 (by VP)
T4: Receive report	T8: Make payment