Applied Informatics University Klagenfurt

ALPEN-ADRIA UNIVERSITÄT KLAGENFURT

Technical Report

2008/001

Institute of Applied Informatics

Intelligent Systems and Business Informatics

June 2008

Model-based repair of web service processes

G. Friedrich, V. Ivanchenko

Model-based repair of web service processes

Abstract.

Orchestrated web services are business processes whose activities are implemented using web services. If the execution of such processes fails due to errors within activities, methods are required to correctly resume workflow instances. This paper follows a model-based approach which identifies faulty activities and generates repair plans so that the faulty workflow instance can finish correctly. Here we focus on the repair aspect, presenting a formalization of the repair problem which is exploited for the computation of repair plans. The result is a model-based repair system for business processes which generates repair plans that guarantee the successful completion of a faulty workflow instance if repair is possible.

1 Introduction

In practical applications of orchestrated web services, failures may occur during the execution of processes, for example when a service provides incorrect output. Currently, designers must implement various handlers at design time to deal with such failures. It was investigated in a research project how models of web service processes can be exploited to diagnose and repair process failures. The aim is reducing programming effort on one hand and providing a more general approach which can deal with all possible fault patterns on the other hand (we avoid citations to allow anonymous reviewing).

Orchestrated web services are specified using process description languages, such as the *Web Services Business Process Execution Language* (WS-BPEL). For generality we assume a simple process description schema and abstract the peculiarities of special languages. Since the root of web service process descriptions is the domain of workflows (WF), we use WF as a synonym for such business processes.

This paper presents the repair part of the mentioned project, focusing on run time failures assuming that the design of the process is correct. The result of our work is a model-based repair system for a typical class of web service processes. Using a WF description, an execution log, and the set of faulty activities provided by a diagnosis system, the system computes a repair plan which guarantees the successful completion of the faulty workflow instance if such a completion is possible.

We start with an example presenting the main ideas and then introduce the properties of the class of processes dealt with. We then determine what does it mean to complete a faulty workflow instance correctly. Based on this correctness property, we define common repair actions, in particular formalizing the (re-)executions of WF activities, the substitution of activities, and compensating for the effects of executed activities. These repair actions are exploited to generate plans which guarantee that all goals of the process are fulfilled after the repair plan was executed. Finally, the method for computing repair plans is described and evaluated showing promising performance results.

2 Example

For the introduction of our approach we use the following example:

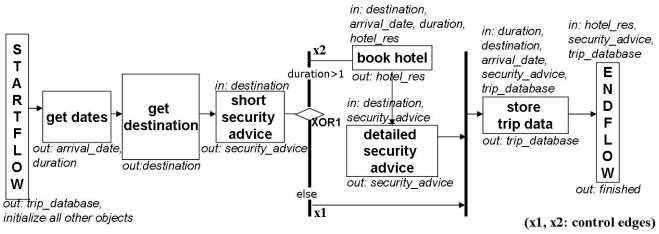

Figure 1. The trip planning example

Let us consider the following WF execution. The input to our example WF is a *trip database*, provided by the activity STARTFLOW which initializes also all other WF objects. An employee inputs trip data and a security agent immediately performs the activity *short security advice* returning standard *security advice*. Since the trip takes longer than a day, a hotel is booked and detailed security advice is generated (which takes some time and contains advice which not normally applicable to day-trips). Then, the trip database is updated. Let us assume an exception is thrown after the trip data was stored (before ENDFLOW is executed) and diagnosis (e.g. applying [1, 5, 9]) indicates that the activity *get dates* is transiently faulty. The outputs (the goals) of the WF are the inputs of ENDFLOW.

Three repair actions are typically employed for dealing with faulty WF instances. Activities can be (re-)executed, substituted, or their effects on objects can be compensated for. Compensating for the effects of the activity *store trip data* on the *trip database* restores the state of the *trip database* before this activity was executed. The goal of *completing* a faulty WF instance is to find a *repair plan* such that after the execution of this plan the output objects [6] of the WF will have correct states. Furthermore, the completion costs should be optimized, for example minimizing the number of repair actions (more sophisticated cost functions are discussed later).

Since it is impossible to predict in general which branches of XOR-splits will be taken at planning time, contingencies must be considered. A plan with a minimal number of actions suggests re-executing *get dates* resulting in correct *duration* and *arrival date* values. Depending on the value of *duration*, the plan considers contingencies: **In case (1)** *duration = 1* (0 days are not possible), the effect of activity *detailed security advice* on *security advice* is compensated for, producing *security advice* as that would have been correctly created by *short security advice*. Following this, the effect of *store trip data* must be compensated for in order to restore the "old" state of the *trip database*. Next, *store trip data* is re-executed, the effect of *book hotel* is compensated for and finally ENDFLOW is executed. Compensating for *book hotel* produces the empty *hotel reservation* which was the initial value written by STARTFLOW. **In case (2)** *duration* > 1, the effect of *book hotel* is compensated for and *book hotel* is re-executed. The activity *detailed security advice* needs no re-execution

because its output was computed correctly. Next, the effect of activity *store trip data* is compensated for and *store trip data* is re-executed, before ENDFLOW is finally executed.

3 Workflow descriptions and semantics

For the formalization of our repair method we briefly present the basic properties of WFs. A WF S is represented by a directed acyclic control graph and a set of objects O [6]. The control graph is defined by a set of nodes representing activities which are connected by control edges. This set of nodes is partitioned into nodes performing some operations and control nodes. Control nodes comprise the types AND-split/AND-join and XOR-split/XOR-join. The control graph includes one start and one end node. Splits are constructed in a block oriented fashion, i.e. exactly one join is associated with each split and all paths starting in a split node join in the associated join node. All input and output objects are specified for each activity a. Output objects of a are the objects effected by a. The input (output) objects of a WF are the output (input) objects of the STARTFLOW (ENDFLOW) node. A correct state of the output objects of a WF is the goal of a WF execution. In order to prevent indeterministic behavior due to parallelism, objects which are effected by an activity $a1$ are not allowed to be read or effected by an activity $a2$ which is executed in parallel to $a1$.

An instance I of a WF S is represented by activity states [6] (e.g. `completed` or `skipped`), the states of the control edges (`notSignaled`, `trueSignaled`, or `falseSignaled`), and the object states (e.g. their value). We denote the state of object o in instance I by $OS^I(o)$.

After initialization, a WF is executed by changing the state of the activities (as the activities are executed), control edges, and objects. The execution of a WF S is a sequence of instances $\langle I_0, \ldots, I_k \rangle$ where each immediate successor I_j of I_i is generated by following the control graph. I_0 (I_{end}) is the initial (end) instance where the execution of the STARTFLOW (ENDFLOW) activity is completed. Execution of a WF is *complete* iff it ends with I_{end}. We assume that ENDFLOW sets the distinct object *finished* to true.

4 Completion of faulty workflow instances

To successfully complete a faulty instance I_c we assume the availability of a set of possible *repair* actions RA. Note, no restriction is made between a repair plan execution and the "normal" execution of a WF since both executions may be interleaved. Consequently, (re-)executed activities are also considered as repair actions.

As indicated in the example, a repair plan of a faulty WF instance I_c of the original WF S can be expressed as a WF.[1] Such a repair plan has to fulfill the property that after the ENDFLOW activity of the repair plan is executed the output objects of the original WF S are in the same state as they would have been after the correct execution of S. Note that WFs as defined above always terminate. More formally:

Definition 1 (Completion) *Given a WF S, an initial instance I_0 of S, the (faulty) instance I_c, a set of repair actions RA, and a repair plan R whose activity nodes are from RA: a repair plan R is a completion of the (faulty) workflow instance I_c iff for all complete executions $\langle I_c, \ldots, I^R_{end} \rangle$ of R a complete execution $\langle I_0, \ldots, I_{end} \rangle$ of the original workflow S exists s.t. for all output objects o of S : $OS^{I_{end}}(o) = OS^{I^R_{end}}(o)$. Note, R starts with instance I_c.*

[1] The example repair plan can be transformed easily to a WF. In principle we could exploit contingency plans for repair instead of WFs, but we want to avoid the introduction of additional concepts.

This definition could be rephrased more generally based on the correctness of object states. We will exploit this correctness property for the generation of repair plans.

Definition 2 (Correctness of object states) *Given a WF S, an initial instance I_0 of S, an instance I_c, and a set of objects $Os \subseteq O$: the object states of Os are correct in I_c iff there exists a WF execution $\langle I_0, \ldots, I_n \rangle$ of S s.t. $OS^{I_c}(o) = OS^{I_n}(o)$ for all $o \in Os$.*

As a consequence, the WF and the inputs to a WF define the correct object states of a WF instance.

Remark 1 *Given a WF S and I_0: a repair plan R is a completion of the instance I_c iff all complete executions of R result in an instance where the set of output objects of S are correct.*

5 Determining correctness of object states

The generation of a repair plan for completing a WF instance has to assess the correctness of object states for various decisions, e.g. in order to decide if object states can be reused for repair. To formally describe our repair approach we employ first-order logic with negation as failure but *without* function symbols. Such logical sentences can be processed by the DLV system [4] which is used in the implementation of the repair system.

A forward branching, acyclic and discrete time representation is used to model time. We follow [2] by modeling the relation of time points using the predicate $next(t1, t2)$. Each time point (except the start time point) has exactly one predecessor and possibly multiple successors. Time points are partially ordered symbols.

We assume the existence of a set of literals $failure(a_i, t_i)$, generated by a diagnosis system, that expresses the fact that the execution of activity a_i at time point t_i was faulty. In addition, the diagnosis system can output $isTransFaulty(a_i, t_i)$ expressing the potentially transient nature of such failures. This, together with the substitution of activities, shows that the correctness of activity executions may change over time. Consequently, the correctness of an activity a_i at a time point t_i is modeled by a binary literal $ok(a_i, t_i)$. The following formalization of the correctness of an activity expresses (in the first clause) that the application of an activity is considered to be incorrect at time point t if diagnosis signals a failure of a at t. We use DLV-notation. The second clause models inertia, i.e. the application of an activity at the next time point remains incorrect unless the activity was substituted or the failure is transient. The third clause models the fact that an activity is assumed to be correct unless evidence to the contrary exists. The literal $node(t)$ expresses that t is a time point.

$-ok(A,T)$ `if` $failure(A,T)$.
$-ok(A,T2)$ `if` $activity(A), -ok(A,T1), next(T1,T2),$ `not` $substituted(A,T2), $`not` $isTransFaulty(A,T1)$.
$ok(A,T2)$ `if` $activity(A), node(T2),$`not` $-ok(A,T2)$.

Since we do not require semantic information about the activities, we have to assume that each application produces a unique object state of the effected objects. Consequently, object states can be thought of as a triple (o, a, v) where o is an object, a is an activity which effected o, and v is the time point of the application of activity a. In our formalization the literal $cversion(o, a, v, t)$ (also called current version) expresses the fact that at time t object o has a state produced by a at time v. In order to avoid overloading the ok predicate, the literal $notsusp(o, a, v, t)$ expresses that the state (o, a, v) of object o is correct (*not susp*ect) at time point t.

The states of control edges are modeled using the states of associated objects. The execution of an XOR results in an assignment of states to these objects. The value of such a state (o, x, v) at time point

t is expressed by the fluent $value(o, x, v, val, t)$ where o is an object associated with an outgoing edge of XOR x, v is the application time point of x, and val is either `trueSignaled`, `falseSignaled`, or `notSignaled`. The execution log provides information about the states of the control edges (i.e. the actual branching).

The correctness of object states can be computed by considering the dependencies of object states produced by the execution of activities. The literal $dep(o_o, a_o, v_o, o_i, a_i, v_i)$ is used to model these dependencies and expresses that the state (o_o, a_o, v_o) of object o_o depends on object state (o_i, a_i, v_i) of an object o_i which was the input to an activity a_o at v_o.

Based on these dependencies conditions can be formulated that must hold if an object state is *assumed to be correct* for a time point. The usual conditions are that the activity which produced an object state, is correct and all its inputs are correct. In addition, correctness of object states at time point t depends on the branching of XOR-splits at t. In our example the activity *detailed security advice* can be executed only if the state of the control edge $x2$ is `trueSignaled` and this value is *correct*. In this case an activity is said to be *enabled*. Let $\mathrm{xEdges}(a)$ be all the objects o associated with control edges on the path from STARTFLOW to activity a where the activation of a depends on the state of these objects o. In our example $\mathrm{xEdges}(\texttt{book\_hotel}) = \{x2\}$ and $\mathrm{xEdges}(\texttt{store\_trip\_data}) = \{\}$. Activity a is *enabled* at t if all xEdges of a are correctly `trueSignaled` at t. This expressed by the literal $enabled(a, t)$.[2]

In addition, the correctness of the results of executing an activity at t depends on the possible activation of other activities. Even if it is assumed that the activity *store trip data* will work correctly, and that this activity is enabled, and that the object states produced by activities *get dates*, *get destination*, and *short security advice* are correct at t (i.e. all input objects of *store trip data* are correct), we cannot conclude that the application of *store trip data* will produce valid results. Only if $x2$ is `falseSignaled` and therefore *detailed security advice* is skipped, *store trip data* produces a valid *trip database*. Consequently, the input objects of an activity a must not be effected by other activities which have to be executed before a. In this case we say that an object has the *right version* for a at t.

The repair of a faulty instance of WF S may require the re-execution of activities. Since semantic information about the implementation of activities is not available in general, the implementation of all activities is assumed to be different. Therefore, an execution of S contains only applications of different activities. Because activities may be indeterministic, re-executions may result in different outputs. Consequently, if during repair an object state was produced by two different applications of an activity, the correctness of this object state can not be guaranteed.

To summarize: An object state (o, a, v) at time point t is considered to be correct if (1) the application v of activity a which produced o was correct at v, (2) at t this activity is enabled (3) at t all inputs of a are correct and these inputs are in the right version for a, and (4) the object state of (o, a, v) was not produced by different applications of the same activities.

After having introduced the concept *"correctness of object states"* we will define the repair actions in the next section.

[2] Note, due to space limitations we cannot give a complete formalization. The complete formalization, including the specification of our example, as well as test cases can be found at http://test-informations.info/

6 Modeling repair actions

In our formalism a repair plan corresponds to a logical model. Consequently, we distinguish between models where an activity a is executed versus models where a is not executed. The execution of activity a at time point t is modeled by the positive literal $do(a, t)$. The fact that a is not executed at t is expressed by the negation $-do(a, t)$.

An activity a can be executed at t if a is *correct* and *enabled* at t. In addition, at t all inputs of a must be correct and must have their right version. As the application of aggregation in our case is currently not possible in DLV, we use negation as failure and the literal $someCurrInputSuspOrFalseVersion(a, t)$ which is true if some inputs of a are incorrect or are of the wrong version at t.

Furthermore, an activity need not be executed if the goal of the repair is reached (literal $goalReached$). For optimization purposes, we constrain execution to certain time points. This is reflected by literal $doAt(a, t)$ which indicates that activity a *can* be executed at t. In addition some time points may become infeasible as explained in the subsequent discussion of the execution of XOR-splits (literal $impossible$). Following [2], the next clause defines the precondition of executing an activity as explained above (`v` is the logical OR):

$$do(A,T) \texttt{ v } -do(A,T) \texttt{ if } node(T), ok(A,T), enabled(A,T), \\ \texttt{not } someCurrInputSuspOrFalseVersion(A,T), \\ \texttt{not } goalReached(T), doAt(A,T), \texttt{not } impossible(T).$$

The effects of the execution of an activity a at time point $t1$ is to add $cversion(o, a, t1, t2)$ literals for all effected objects of a where $t2$ is a direct successor time point of $t1$. The $cversion$ literal is inertial [2]. As long as the negation of a $cversion$ literal cannot be deduced for a time point $t2$ and this literal is true in $t1$, the truth of this literal is assumed at $t2$. Interia is canceled for the $cversion$ literals of the effected objects o of activity a (defined by facts $ineffset(a, o)$) by deducing their negation:

$$cversion(Oo, A, T1, T2) \texttt{ if } do(A, T1), next(T1, T2), \\ ineffset(A, Oo).$$
$$-cversion(Oo, AX, VX, T2) \texttt{ if } do(A, T1), next(T1, T2), \\ ineffset(A, Oo), cversion(Oo, AX, VX, T1).$$

Furthermore, the effect of action $do(a, t)$ is to add all $dep(o_o, a, t, o_i, a_i, v_i)$ literals where o_o is an effected object of a and (o_i, a_i, v_i) is the object state of an input object of a. In order to perform compensation actions the previous states of objects must be recorded before they were effected by activity a. Literal $vbe(a, t, o, a2, t2)$ (*v*ersion *b*efore *e*xecution) expresses the fact that at time t before activity a effected object o, object o had state $(o, a2, t2)$, i.e. o was produced by $a2$ at $t2$.

Since the effects of XOR-splits may not be known at planning time on one hand and different repair actions are needed for different branches of an XOR-split on the other hand, we consider two successor time points for each XOR-split. In these time points the objects representing the state of the control edges of an XOR receive different values. E.g. if we simulate the execution of `XOR1` at planning time then there are two possible successor time points. In one time point the state of $x1$ is `trueSignaled`. In the other time point $x2$ is `trueSignaled`. In case an XOR-split x was already executed before planning at t, and at planning time the re-execution of x has the same inputs as at t then we can reuse the observed outcome of x. Consequently, it is known at planning time which branch of x will be `trueSignaled` and which branch of the XOR-split becomes *impossible* (i.e. it will be never followed). E.g. if in our example *duration* would have been correctly determined to be 1 day then the case where *duration* is greater than 1 day need not be considered.

Another repair action is the *substitution* of activities, more precisely the replacement of the implementation of an activity with another that provides the same functionality. For example we could change the provider of security advices. Substitution is modeled by the following clauses:

$subs(A,T)$ `v` $-$ $subs(A,T)$ `if` $subsAt(A,T)$, `not` $impossible(T)$.
$substituted(A,T2)$ `if` $subs(A,T1)$, $next(T1,T2)$.

Similar to *do*-actions we restrict the applicability of substitutions to certain time points using the $subsAt$ literal. The effect of a substitution is the assertion of $substituted(a,t)$ which blocks the inertia of $-ok(a,t)$ and therefore results in the assertion of $ok(a,t)$. Note, some services may be provided by the same agent and may have some private objects (not accessible by other agents). Substituting an agent can be modeled by substituting all activities of the agent and canceling the availability of private objects.

A further repair action supported by our system is compensation for the effects of activities. Since our target application area is web (information) services, we assume that effects of activities may be *compensated for* on an individual basis. However, multiple compensations may be required to assure the correctness of object states. We distinguish two different compensation actions. Some compensation actions do not need any input, e.g. the compensation of *store trip data* writing the *trip database* could be implemented by saving the *trip database* before it is written. This type of compensation action is termed *always-compensateable*. An alternative implementation could simply record the changes performed by *store trip data*. In this case a compensation action requires the state of the *trip database* right after *store trip data* effected the *trip database*. We term this type of compensations as *one-step-back* compensations. The following clause formalizes the *always-compensateable* action. We assume that if an activity is re-executed several times, only the effects of the last execution can be compensated for. If the provider of an activity offers more sophisticated compensation techniques then this assumption could be dropped.

$comp(A,V,O,T)$ `v` $-$ $comp(A,V,O,T)$ `if`
$lastCompleted(A,V,T)$, $alwaysComp(A,O)$,
$compAt(A,O,T)$, `not` $impossible(T)$.

The effect on object o of an activity a applied at time point v can be compensated for at time point t if (1) the application of a at v was the last completed execution of a, (2) the compensation of object o effected by a can be executed without any input, and (3) the compensation action is allowed at time point t and this time point is valid. The modeling of the *one-step-back* compensations requires one addition literal which is true if the input object for compensation is in the right state. The effects of a compensation action is formalized by the following two clauses:

$cversion(O,Ai,Vi,T2)$ `if` $comp(A,V,O,T1)$,
$next(T1,T2)$, $vbe(A,V,O,Ai,Vi)$.
$-cversion(O,Ax,Vx,T2)$ `if` $comp(A,V,O,T1)$, $next(T1,T2)$,
$cversion(O,Ax,Vx,T1)$.

The first clause specifies that after the compensation of object o effected by a at time point v, the object has the same state as it had immediately before a affected it. The second clause cancels inertia by asserting the fact that the state of a compensated object at $t1$ is not an object state at $t2$, where $t2$ is an immediate successor of $t1$.

7 Time structure

In order to avoid searching all possible forward time branch structures, we generate a time skeleton in form of a directed tree of time points. Let us initially assume that only *(re)do*-actions are available. Furthermore, we eliminate AND-split/AND-joins of a WF S by transforming parallel paths into sequential paths. This does not change the set of possible results of a workflow since the parallel executions are assumed to be independent.

The generation of a time skeleton is based on the fact that a sequence of *do*-actions, which is required for the completion of a faulty instance, is a subsequence of a path from STARTFLOW to ENDFLOW of S. If there is no such a subsequence then adding or changing the order of *do*-actions cannot result in a sequence where the correctness of WF output objects can be guaranteed since no description of the activities' functionality is available. Consequently, the WF S is unfolded into a *directed tree* S', i.e. the result is a directed tree with the root STARTFLOW and leaves marked with ENDFLOW where each path from STARTFLOW to ENDFLOW in S' corresponds to a path from STARTFLOW to ENDFLOW in the original WF S. The structure of S' is reflected by $next(t1,t2)$ facts. Nodes of S' correspond to time points and are marked with the activities of S such that the execution order is preserved. XOR-joins are deleted. Every node marked with an activity has exactly one successor time point, except XOR-splits which have exactly two successor time points. An additional *final* time point follows each ENDFLOW because the execution of an activity requires a successor time point. S' is known as the repair plan. Note S' can be extended into a WF according to our definition. For each activity a which might be (re-)executable, $doAt(a,t)$ is added where t corresponds to the time point in S' and t is marked with a. Some activities, however, might not be re-executable either by definition (i.e. STARTFLOW) or by design.

Furthermore, executed activities must be modeled. For each executed activity a time point is generated and *do* and *next* facts are asserted. The root of S' is the direct successor of the last executed activity of the execution log. In our example the asserted facts are $\{do$(STARTFLOW,1), $next$(1,2), do(get_dates,2),. . .,do(short_security_advice,4),. . ., do(detailed_security_advice,7), $next$(7,8), do(store_trip_data,8), $next$(8,9), $next$(9,10), $doAt$(get_dates,10),. . ., $doAt$(XOR1,13), $next$(13,14), $next$(13,17), $doAt$(store_trip_data,14), $doAt$(book_hotel,17), . . .$\}$. These facts define the time points of *possible* activity executions. The path starting at time point 14 (17) corresponds to the case where *duration* is equal to (greater than) 1 day. The activity *store trip data* is contained in both paths.

The insertion of time points for possible compensation actions is based on the fact that compensation restores an object o to the state it was before activity a. We insert time points before activities where the object state produced by a compensation action can be used as an input. E.g. compensating the effect of *detailed security advice* executed at 7 on object *security advice* produces a state of *security advice* as it was after the execution of *short security advice* at 4. This object state is the input to *store trip data* possibly executed at 14 (case *duration*=1). Consequently, we insert time points and $compAt$ literals for all possible compensation actions (some compensation actions could be unavailable) where the effect of such compensation can be exploited by a subsequent activity. Note, this must be done iteratively since in the case of *one-step-back compensation*, the compensation actions themselves may require preceding compensation actions. An important property of these compensation actions is that they restore just the state of an objects which is needed by a direct successor action. No other objects are effected. Consequently, a compensation action does not prevent any other action from being executed and the completeness of the approach is preserved.

In a final expansion step of S', $subsAt$ literals are added before

permanently faulty activities. In case such an activity must be executed, its implementation is substituted. Again no other action is prevented from executing.

8 Computation and evaluation

A repair plan is a subset of a stable model generated by the DLV system. In order to guarantee that the output objects of a repair plan are correct in all models, we ensure that repair plans guarantee the completion of a faulty WF instance using the following clauses:

$$goalReached(T) \texttt{ if } cversion(finished, endflow, V, T),$$
$$notsusp(finished, endflow, V, T).$$
$$\texttt{false if } finalNode(T), \texttt{not } goalReached(T), \texttt{not } impossible(T).$$

The goals of a WF are fulfilled in time point t if the object *finished* is correct (first clause). The second clause eliminates all logical models for which a final time point exists where the goals of the WF are not fulfilled.

Within the terminology of planning, every logical model contains a repair plan which is both conditional and secure (i.e. its success is guaranteed). The input to the DLV system is a knowledge base which contains the formalization of the repair problem, a description of the WF comprising of information about which effects of activities can be compensated for (including the type of compensation) and which activities are substitutable. Furthermore, the time structure is added as explained above and information about which actions may be executed at which time points. Finally, the execution log (including information about the output of XOR-splits) and failure descriptions are included in the input.

DLV offers the modeling of costs using soft constraints. A simple strategy is to minimize the overall number of actions. A more sophisticated implementation takes branching probabilities and the different costs of *compensations*, *substitutions*, and *(re-)executions* into account, but this is beyond the scope of this paper.

We evaluated our approach by testing it on a series of more than 1000 randomly generated WFs. The generation was parameterized by the number of activities, the number of objects, and the number of XOR-split points. We did not consider AND-split/AND-joins; such WFs can be reduced to sequential WFs. Stopping the WF execution because a fault was discovered and the assignment of failures to activities was also randomly chosen. In order to increase the search space all activities were assumed to be one-step-back compensateable and substitutable.

In addition, a functionality for activities was randomly generated. This functionality was exploited to test the correctness of our repair plans. In particular repair plans were started following a faulty instance and the state of the output objects of the WF was computed for every possible path of the repair plan. We checked if the state of the output objects of these paths could be computed by executing the corresponding part of the original WF where no failures occurred. Correct repair plans were subsequently computed for all tests. The computation time required for WFs containing 45 activities, 45 objects, at most 3 failures, and 9 XOR-splits was on average 45 sec (using a 2x Intel Xeon CPU 3.00GHz (Single Core), 4GB RAM machine) when the number of actions was minimized. Consequently, the approach is feasible for the medium sized orchestrated web service used in practice (test cases are published at http://test-informations.info/).

9 Related work

Our work can be classified as conditional planning with background theories. In [2] planning with background theories based on the DLV-system was introduced and subsequently extended to conditional planning by [8] where optimistic, possibly insecure, plans are generated. Consequently, we have extended these ideas to facilitate the generation of plans that guarantee successful completion.

In the area of web service composition, many papers have been published about exploiting planning techniques. An excellent overview is given by [3]. To sum up, on one hand current planning techniques which allow sufficiently expressive background theories do not support conditional planning, and on the other hand current conditional planning techniques strongly limit the expressivity of the background theory and plan optimization. We have chosen to use a very expressive reasoning system in order to formalize conditional planning for the presented application domain.

The proposals for repairing faulty WF instances found in the WF literature focus on methods to support the design of exception handlers. E.g. in [7], WF exception patterns are classified and analyzed. Based on these investigations a graphical language is presented which allows exception handling strategies to be defined. Our approach differs significantly, since we exploit the WF definition, the execution log, and the provided failure information. Using this information, we can formulate a search process which exhaustively takes all possible repair plans into account. Exception handling based on repair actions is provided automatically.

10 Conclusions

In this paper we have proposed a method for generating repair plans for faulty WF instances. The repair plan exploits popular repair strategies such as compensating for the effects of activities, and substituting and (re-)executing activities. Our approach is based on a logical formalization which can be processed by modern logical reasoning systems supporting disjunctive logic programming. Furthermore, we conducted an extensive evaluation on a random set of repair problems. The result is a model-based repair system that generates conditional plans which guarantee the successful completion of faulty workflow instances.

REFERENCES

[1] L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, M. Segnan, and D. Theseider Dupré, 'Enhancing web services with diagnostic capabilities', in *European Conference on Web Services*, pp. 182–191. IEEE Computer Society, (2005).

[2] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres, 'A logic programming approach to knowledge-state planning: Semantics and complexity', *ACM Trans. Comput. Log.*, **5**(2), 206–263, (2004).

[3] J. Hoffmann, P. Bertoli, and M. Pistore, 'Web service composition as planning, revisited: In between background theories and initial state uncertainty', in *AAAI*, pp. 1013–1018, (2007).

[4] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, 'The dlv system for knowledge representation and reasoning', *ACM Trans. Comput. Log.*, **7**(3), 499–562, (2006).

[5] W. Mayer and M. Stumptner, 'Debugging failures in web services coordination', in *SEKE*, pp. 536–543, (2006).

[6] M. Reichert and P. Dadam, 'Adept$_{\text{flex}}$-supporting dynamic changes of workflows without losing control', *J. Intell. Inf. Syst.*, **10**(2), (1998).

[7] N. Russell, W. Aalst, and A. Hofstede, 'Workflow exception patterns', in *CAiSE*, volume 4001 of *LNCS*, pp. 288–302. Springer, (2006).

[8] D. van Nieuwenborgh, T. Eiter, and D. Vermeir, 'Conditional planning with external functions', in *LPNMR*, volume 4483 of *LNCS*, pp. 214–227. Springer, (2007).

[9] Y. Yan and P. Dague, 'Modeling and diagnosing orchestrated web service processes', in *ICWS*, pp. 51–59, (2007).