

Reasoning about Repairability of Workflows at Design Time

Gaston Tagni, Annette ten Teije, and Frank van Harmelen **

Department of Artificial Intelligence
Vrije Universiteit Amsterdam, The Netherlands,
De Boelelaan 1081a, 1081 HV Amsterdam
{gtagni, annette, Frank.van.Harmelen}@cs.vu.nl

Abstract. This paper describes an approach for reasoning about the repairability of workflows at design time. We propose a *heuristic-based analysis* of a workflow that aims at evaluating its definition, considering different design aspects and characteristics that affect its repairability (called *repairability factors*), in order to determine if the workflow schema supports repairable executions of its activities through the application of repair actions. The analysis intends to identify and evaluate the impact of critical design flaws affecting the repairability of workflows. The results of this analysis are fed back to the workflow designer and used to improve the repairability of the workflow by making appropriate changes to its definition.

Key words: Web Service Composition, Repairability of Workflows, Self-healing Web services, Workflow Design

1 Introduction

Composition of Web services, or service integration, is an area of active research that focuses on the aggregation or combination of Web services with the potential to enable Business-to-Business (B2B) or Enterprise Application Integration (EAI). As an example, consider an e-commerce application responsible for selling and delivering goods to its customers. Such an application can be built based on the interaction of multiple partner services such as warehouses, suppliers and on-line shops. These constituent services interact with each other in order to achieve a common business goal, in this case, the efficient delivery of goods to its customers. Such a composition is typically described by a workflow where primitive activities denote invocations to Web service operations and structured activities implement control-flow dependencies between activities. We call such workflows *Web service-based workflows* because automated activities are implemented by using the functionality provided by Web services.

One of the desired characteristics of a composition of Web services is its *repairability*. Intuitively, this concept refers to the ability of a system to cope with any unexpected anomalous situations [3]. Repairability is also related to the notion of *fault tolerance*, which is understood as the ability of a system to provide the expected functionality even under the occurrence of failures. To some

** Research funded by EU IST FET-STREP n.516933 WS-Diamond

extent, repairability is a design issue, in the sense that it requires the workflow definition to support the execution of corrective actions (also known as repair actions) upon failures that have been detected at runtime. This in turn calls for methodologies and tools to support the design and evaluation of workflow specifications that enable corrective measures at runtime.

Several approaches have been proposed that consider the reparability of workflows, as well as other important properties, at different stages of the workflow development cycle. The application of Petri nets to validation of workflow definitions has been extensively studied in [1, 2, 10]. The work done in [5] explores the integration of transactional features and Workflow Management Systems (WfMS) to support workflow recovery at runtime. Some approaches exist [6, 13] that consider workflow recovery at design time based on the concept of exception handling. For example, in [6] the authors focus on the combination of exception handling mechanisms and the transaction concept used in databases to support the design and implementation of reliable workflow processes. They describe an algorithm to validate the correctness of a workflow to ensure that workflow executions do not reach unrecoverable states. Workflow recovery and adaptability have also been studied in [11, 12]. To the best of our knowledge, however, none of the current approaches considers evaluating the correctness of a workflow schema to ensure that repair actions can be applied at runtime.

Motivated by the need to support the design of conversationally complex composite Web services, this paper describes an approach for reasoning about the repairability of workflows at design time that aims at assisting designers in the definition of repairable workflows. More specifically, we propose a *heuristic-based analysis* of a workflow that aims at evaluating its definition, considering different design aspects and characteristics that affect its repairability (called *repairability factors*), in order to determine if the workflow schema supports repairable executions of its activities through the application of repair actions. In other words, the method intends to identify and evaluate the impact of critical design flaws affecting the repairability of workflows. The results of this analysis are fed back to the workflow designer and used to improve the repairability of the workflow by making appropriate changes to its definition.

The paper is organized as follows: Section 2 introduces some preliminary information and some of the assumptions the approach is based on. In section 3, the notion of workflow repairability is introduced. Section 4 describes the heuristic-based approach proposed, defines the heuristic function to be used and discuss the repairability factors considered by the reasoning method. After introducing the theoretical background we proceed to describe in section 5 the algorithm implementing the heuristic-based analysis. Section 6 presents the experimental results. Finally, section 7 reports the conclusions and future work.

2 Preliminaries

In this work we assume the workflow designer is able to provide information about failure and branching probabilities associated to the activities in the workflow. Moreover, the repair model being considered is based on the execution of a sequence of repair actions. These include: *substitute*, *compensate* and *retry*. The semantics of *substitute*(A, B) is that activity A (its service provider) is replaced

by the *functionally equivalent* activity (another service provider) B . We assume the designer is able to determine whether two activities are functionally equivalent. As for $compensate(A, B)$, it means that activity A is compensated by activity B , i.e. the execution of B rolls back the effects caused by the execution of A . Finally, $retry(A)$ means that activity A is re-executed. In the following we introduce the workflow and execution models used throughout the paper.

2.1 Workflow Model

For modeling workflows we decided to use a *graph-based approach* similar to the one proposed in [7] and represent workflows as directed graphs. The main reason for this was the simplicity and adequacy of this model to support the type of analysis we are interested in. More specifically, a workflow schema is modeled as a directed graph, called the *control graph*, which captures the behavioral and functional aspects of a workflow specification. Nodes in the graph denote activities while directed arcs (called *control arcs*) represent interconnections between activities. Nodes are classified into *activity nodes* and *control nodes*. The former model primitive activities, which include activities that use the functionality provided by some partner service (called *external activities*) and those activities used for implementing the data flow as well as other aspects of a workflow business logic, e.g. exception handling. Control nodes, instead model control-flow structures (*structured activities*). We consider workflows expressed using the five *basic control-flow* patterns along with the *structured loop* and *multi-choice* patterns described in [8]. Finally, data dependencies between primitive activities are modeled using a directed graph (called *data graph*) that specifies the input and output objects of each activity.

From all the variables defined in the workflow we consider the subset of goal variables. A *goal variable* is a variable whose value can be used for evaluating the correctness of a workflow execution. From this set it is also possible to compute the set of *goal-dependent activities*. Such activities modify a goal variable either directly or indirectly through a chain of data dependencies with other activities. Goal-dependent activities are important for repairability purposes since their repairability affect the correctness of a workflow execution.

Example 1. Consider a workflow W_1 for planning a business trip. Activities in W_1 allow the users to purchase flight tickets, make hotel and dinner reservations (if the duration of the trip is more than 1 day) and, get the weather forecast for the destination. Activity $A_1 : bookFlight$ takes as input the initial budget (variable *Budget*) and *CityName* and gives as output the *ETicket*, *ArrivalTime*, *DepartureTime* and the new *Budget*. Activity $A_2 : bookHotel$ takes *CityName*, *TripDuration* and *Budget* as input and outputs the new *Budget* and the *HotelName*. Activity $A_3 : reserveDinner$ uses *CityName* as input and outputs the *Restaurant*. Finally, the input of activity $A_4 : getWeatherReport$ is *CityName* while its output is *WeatherCondition*. Figure 1 depicts the resulting workflow. Goal variables are specified between brackets.

2.2 Workflow Execution Model

The execution of a workflow schema S is modeled as a sequence $\langle I_0, \dots, I_n \rangle$ where each I_i ($1 \leq i \leq n$) is an instance of S . Intuitively, a workflow instance

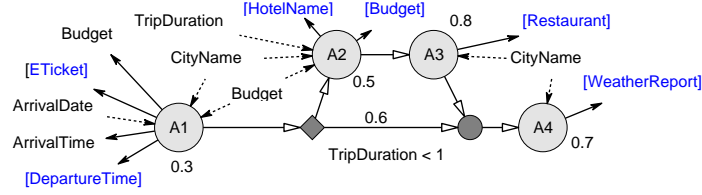


Fig. 1. Trip-planner Workflow

captures the state of the workflow at a given point during its execution. That is, an instance specifies the state of every activity in the workflow (*not activated*, *activated*, *running*, *skipped* and *completed*), the state of every arc defined between activities (*not signaled*, *signaled*) and the state of every data object specified in the workflow (*ok*, *suspected*, *unsafe*). The state *unsafe* means that the value of an object is incorrect while *suspected* state means that its value might be incorrect, e.g. due to data dependencies with other *unsafe* or *suspected* objects.

3 Repairability of Workflows

During workflow execution, a WfMS is responsible for executing workflow activities according to the control flow specified by the designer. The WfMS is also responsible for monitoring the execution of activities with the aim of detecting faults that may occur during or after the execution of each activity. When a fault is detected the workflow execution stops and a diagnosis process is started in order to discover the causes of the faults. The result of the diagnosis task is the current (faulty) workflow instance and a set of *abnormal activities*; i.e. functionally incorrect activities. A repair module uses the information provided by the diagnosis module in order to generate a combination of repair actions such that when executed it can repair the faulty instance and resume the correct execution of the workflow.

From this, it is possible to see that the repairability of workflows depends on the ability to repair faulty instances. Then, the repairability of workflow instances is defined as follows:

Def. 1 (Repairability of faulty instances) *A faulty instance I_f is repairable iff for every activity that needs to be repaired (abnormal and infected activities) there exist a repair.*

In this definition, *repair* means that it is possible to either execute a set of predefined repair handlers or find a combination of repair actions such that, when applied to the faulty instance, brings the workflow execution to a correct final state. We call the first approach, *predefined repair* and the second approach, which is based on the computation and execution of a combination of repair actions at runtime, *automatic repair*. In contrast to abnormal activities, *infected activities* are those that, although not abnormal, they read data written by other abnormal or infected activities. Next we define repairability of an activity.

Def. 2 (Repairability of an activity) *Let S be a workflow schema and A an external activity of S . Then, A is repairable iff there exist a set of repair actions or predefined repair handlers that can be applied to A .*

Based on these definitions we introduce now the notion of repairability of a workflow schema, which defines what it means for a workflow schema to be repairable.

Def. 3 (Weak Repairability of Workflow Schemas) *A WF schema S is repairable iff there exist at least one faulty instance I_f of S ($I_f(S)$) such that I_f can be repaired.*

Although it is not feasible to enumerate all possible (faulty) instances of a workflow schema and compute their repairability, it is however possible to estimate the repairability of a workflow using heuristic information. Notice that this definition imposes only a very weak requirement on repairability (that is why we use the term *weak repairability*). The other option is to require that all faulty instances can be repaired, which would be referred to as *strong repairability*. However, this would be far too strong a requirement.

4 Heuristic-based Repairability Analysis

With the goal of determining the repairability of a workflow schema we propose to perform a *heuristic-based analysis* of the workflow definition in order to know whether the schema supports repairable executions of its activities. Such analysis considers various *repairability factors* that influence the repairability of activities and of the entire workflow. Intuitively, a repairability factor can be understood as any characteristic or attribute of a workflow design that influences the possibility of repairing faulty executions. For example, the definition of predefined fault handlers associated to an activity is a factor affecting its repairability.

The method proposed here inspects a workflow definition with the goal of identifying critical design flaws affecting the repairability of activities and, evaluating the impact of such flaws on the repairability of the entire workflow. The results of this analysis are fed back to the workflow designer and constitute valuable information that can be used in order to devise changes to the schema that can improve the repairability of the workflow. More specifically, the outcome of this analysis is a set of non-repairable activities, each of which is enriched with information about different workflow characteristics affecting its repairability. For example, the analysis could tell the designer that an activity is not repairable because no predefined repair handlers have been associated to it. In addition to this, the repairability reasoner evaluates the impact each non-repairable activity on the repairability of the workflow and ranks activities based on their relative importance. The ranking of activities helps designers to focus their attention on those activities whose repairability affects the overall repairability the most.

4.1 A Heuristic for Reasoning about Repairability

From the definitions introduced above it is possible to see that, if every abnormal and infected activity in a faulty instance I_f is repairable then I_f is repairable and,

in general, if every activity in I_f is repairable so is I_f . In the same way, if none of the activities in I_f are repairable then I_f cannot be repaired. Consequently, *the probability of a faulty instance I_f being repairable varies according to the number of repairable activities*. More specifically, by increasing the number of repairable activities the probability of I_f being repairable does not decrease. In the same way, by decreasing the number of repairable activities in I_f the probability of I_f being repairable does not increase.

In the same way that the number of repairable activities is related to the repairability of workflow instances, the repairability of a workflow schema is related to the number of repairable activities. Based on this relation we can define the *heuristic function* $h(s)$ as follows:

$$h(s) = \text{number of repairable activities of workflow schema } s.$$

On the one hand, $h(s) = 0$ means that none of the activities are repairable hence the workflow is not repairable. On the other hand, $h(s) = N$ (where N is the maximum number of external activities) means that every activity in the workflow is repairable and therefore every faulty instance of the workflow can be repaired.

4.2 Repairability Factors

In the following we discuss the repairability factors that we have considered for the repairability analysis of workflows.

- *Repairability of constituent activities*: By definition this is the most important factor affecting the repairability of workflow schemas.
- *Set of repair actions*: The expressivity of a set of repair actions plays an important role in the assessment of a workflow’s repairability as it influences the ability of a repair module to recover the workflow execution from different types of faults.
- *Data and control-flow dependencies*: Data and control-flow dependencies between activities are central to both workflow repair and the repairability analysis of workflows for two important reasons. First of all, they are used for computing the set of goal-dependent activities. Second of all, the impact of non-repairable activities on the repairability of the entire workflow varies depending on the control-flow dependencies between activities.
- *Predefined repair handlers*: Workflow languages usually allows designers to associate arbitrarily complex fault/compensation handlers to groups of activities. Such handlers are used by the exception handling mechanism provided by the WfMS in order to repair user-defined, application specific errors. Therefore, their repairability influences the repairability of the activities they are associated to. Notice, however, that the absence of repair handlers does not necessarily mean that an activity cannot be repaired as a combination of repair actions may be applicable.
- *Equivalent and compensating activities*: In order for $substitute(X)$ to be applicable there must exist at least one functionally equivalent activity that can be used to replace X for. In the same way, for the $compensate(X)$ action to be applicable there must be at least one compensating activity that can be

used to rollback the effects caused by the execution of X . Equivalent and compensating activities can be provided by the workflow designer based on the analysis of different aspects such as QoS agreements and security. The lack of such information can be interpreted as a design flaw affecting the repairability of activities.

- *Workflow instance and the recovery process*: The repairability of workflows also depends on *instance data*, i.e. the state of the faulty workflow execution. Moreover, the computation of a sequence of repair actions depends on the state of the recovery process, e.g. it depends on the effects caused by other recovery actions.
- *Type of Web service operations*: As pointed out in [4], the enforceability of repair actions depends not only on the type of fault detected but also, on the type of Web service operations used by activities in the workflow. For example, using the conversation models described in [4] it is possible to know whether a partner service provides the means for compensating activities, i.e. whether an operation is compensatable.

5 Repairability Reasoning Algorithm

This section describes the algorithm for reasoning about the repairability of workflows, which is implemented by a Workflow Repairability Reasoner (WfRR). It implements the heuristic-based approach proposed in section 4 and builds upon the definitions introduced in the previous sections. The algorithm also takes into account the repairability factors discussed in section 4 to define rules for checking the repairability of activities and for determining the impact of non-repairable activities.

The input of the WfRR is a workflow definition extended with repairability information provided by the workflow designer. This information includes failure and branching probabilities for activities, the list of compensating and equivalent activities, the list of predefined fault/compensation handlers and the list of goal variables. In addition to this, the reasoner uses a model of the conversations supported by the workflow with every partner service. As for the output, the repairability reasoner returns a list of non-repairable activities ranked according to their relative impact on the repairability of the workflow. For each such activity the reasoner specifies several design characteristics affecting its repairability.

Starting from the workflow definition the WfRR processes the workflow schema by traversing its control graph and processing every constituent activity according to its type. For every external activity the WfRR *computes its goal dependency property* in order to determine whether the activity is goal-dependent. If the activity is not goal-dependent it is discarded and the reasoner processes the next activity. Otherwise, the WfRR calls an *Activity Repairability Reasoner* which is responsible for determining the repairability of constituent activities based on multiple repairability factors. After all non-repairable activities have been identified the WfRR *evaluates their impact on the repairability of the workflow*. This is done by considering failure and branching probabilities as well as the control-flow structure of the workflow. The outcome of this step is a ranked list of non-repairable activities that reflects the relative importance of each of them with respect to the repairability of the workflow.

5.1 Computing Goal-dependent Activities

The goal dependency property of an activity depends on the particular execution path followed by each workflow execution. This is because the execution of control-flow structures such as XOR-splits determines the set of activities to be executed next and hence, it affects the actual data dependencies between activities. Therefore, the lack of instance data makes the computation of this property at design time more difficult and less precise than computing it at run-time. Although under certain circumstances it is possible to precisely determine an activity's goal-dependency, e.g. when the conditions of XOR-split activities are known a priori, in general, this is not possible. Consequently, the algorithm associates with every activity its probability of being goal-dependent, which is determined based on branching probabilities.

5.2 Checking the Repairability of Activities

Checking the repairability of an activity involves two tasks. The first one consists in *checking the repairability of predefined repair handlers* associated to the activity, while the second one consists in *checking the applicability of repair actions*. On the one hand, if the workflow definition is such that neither predefined repair handlers nor a combination of repair actions can be applied then the activity is not repairable. On the other hand, if either predefined handlers or a combination of repair actions (or both) can be applied then the activity is repairable. In the following, we describe in more details how these two steps are implemented.

Checking Applicability of Repair Actions The applicability of repair actions depends on the workflow instance being repaired, the state of the repair process, the state of the conversations supported by partner services, the workflow definition and the diagnosability of faults, among others. At design time, however, the lack of instance data makes the process of assessing the applicability of repair actions less precise.

The method proposed here consists in verifying whether the preconditions for the applicability of each repair action hold w.r.t the workflow schema. For example, in order for the *substitute* repair action to be applicable the workflow definition must be such that the activity has at least one equivalent activity associated to it. Due to incomplete information these constitute *necessary but not sufficient* conditions for the applicability of repair actions, i.e. their unsatisfiability implies the non applicability of the repair action whereas their satisfiability does not guarantee that the enforceability of the repair actions at repair time.

Furthermore, the algorithm uses the model of the conversations supported by each partner of the workflow [4] in order to know the type of operations provided by each partner service and thus refine the preconditions of repair actions. In particular, operations provided by Web services can be classified as *retrievable*, if it has no effects on neither the workflow nor the service, i.e. it can be retried multiple times, *compensatable*, if the service provides the means for compensating its effects and, *permanent*, if the operation is neither compensatable nor retrievable. This information must be provided by the designer of the services used by the workflow and therefore it is available at design time. For the purposes of implementing the algorithm we decided to model this information using a XML

document that contains the value of the properties mentioned above for each operation (activity). In the following we define the rules for every repair action together with a possible sequence of actions to repair an activity:

- Rule 1: $retry(A)$
 1. if $reliable(A)$ then $[retry(A)]$
 2. if $compensatable(A) \wedge \exists B : compensating(A, B)$ then $[comp(A), retry(A)]$
- Rule 2: $compensate(A)$
 1. if $compensatable(A) \wedge \exists B : compensating(A, B)$ then $[comp(A), retry(A)]$
- Rule 3: $substitute(A)$
 1. if $reliable(A) \wedge \exists B : equivalent(A, B)$ then $[subs(A), retry(A)]$
 2. if $compensatable(A) \wedge \exists B : compensating(A, B) \wedge \exists C : equivalent(A, C)$ then $[comp(A), subs(A)]$

Note that rule 2 concludes $retry(A)$ because a compensatable activity can always be retried provided its effects are undone first. In the same way, rule 3.1 concludes $retry(A)$ because A is reliable. Finally, rule 3.2 concludes $subs(A)$ because an activity can be substituted provided its effects are compensated first and an equivalent activity exists.

Checking Repairability of Predefined Handlers The next step consists in verifying if the predefined handlers attached to an activity support repairable executions of it. For this, we use the same ideas discussed in [6] and consider the characteristics of repair handlers during the validation of the workflow definition complemented with the conversation models mentioned before. The algorithm assumes that a finite, loop-free list of predefined handlers can be computed statically from the workflow schema. It also assumes that handlers are primitive activities, i.e. no structured activities or sub workflows are supported (although extending the algorithm with such a recursive step would be straightforward). The algorithm returns the list of non-repairable handlers. More precisely, it checks that handlers associated to reliable activities are also reliable and, those associated to compensatable activities are compensatable as well. Moreover, handlers attached to non-reliable activities must be reliable and those associated to activities placed after permanent activities must be reliable.

5.3 Evaluate Impact of non-repairable Activities

After the WfRR computes the set of non-repairable activities, the final step of the algorithm is to evaluate the extent to which non-repairable activities affect the repairability of the workflow. The result of this evaluation is a list of non-repairable activities ranked according to their relative impact. The impact of each non-repairable activity can be measured by considering different workflow characteristics such as the branching and failure probabilities, QoS parameters associated to the activities, the complexity of each activity or their type, etc..

The approach used here considers a combination of the failure and branching probabilities of an activity. More precisely, we associate with every non-repairable activity a *relevance index* which captures the relative importance of the activity. Such index is defined as the product between the branching probability and the failure probability of an activity. That is, given a non-repairable

activity A whose branching and failure probabilities are $bProb(A)$ and $fProb(A)$ respectively, the relevance index is defined as:

$$ri(A) = bProb(A) \times fProb(A) \quad (1)$$

This index imposes an ordering on the list of non-repairable activities that satisfies the following conditions:

- non-repairable activities with branching probability $bProb = 0$ are discarded. This is a desire property since such activities are never executed and therefore their repairability does not influence the repairability of the workflow.
- non-repairable activities whose failure probability is $fProb = 0$ are discarded. The same reasoning as before applies here although in this case activities are executed but never fail.

The evaluation of the impact of non-repairable activities can be improved by considering certain *loop constraints*. For example, the relevance index could be extended to consider the expected maximum number of iterations of an activity inside a loop along with their associated probability distributions. Naturally, this would require the workflow designer to provide information about probability distribution for loops.

5.4 An Example

Consider the workflow illustrated in Figure 1. The workflow schema includes the specification of failure and branching probabilities. Let us assume that the model of the conversations and the repairability information provided by the workflow designer is as follows:

- *equivalentActivity*(A_2 , *reserveHotel*)
- *repairHandler*(A_4 , *defaultWeatherService*)
- *compensatingActivity*(A_3 , *undoReservation*)
- *retrievable*(A_2), *retrievable*(A_4), *compensatable*(A_1), *compensatable*(A_3)

The algorithm starts by processing activity A_1 . Initially the list of non-repairable activities is empty. The first thing the algorithm does is to check if A_1 is goal-dependent. In this case activity *bookFlight* is goal-dependent since it writes the goal variable *DepartureTime*. Then, the algorithm checks whether A_1 is repairable. In this case, the list of predefined repair handlers for the activity is empty hence the algorithm checks the applicability of repair actions. Following the rules described in the previous section, it is possible to see that none of the repair actions can be applied to A_1 . The reason is that although according to the conversation model the activity is compensatable, no compensating activity has been associated to it and thus it is not possible to compensate A_1 . Neither is it possible to retry or substitute it. Since no predefined repair handlers is associated to A_1 and no repair action can be applied the algorithm concludes that A_1 is not repairable and adds it to the list *nonRepairableActivities*. Following the same approach it is possible to see that A_2 , A_3 and A_4 are repairable. Applying the rules introduced in the previous section it is possible to conclude that A_2 could be repaired by executing either [*retry*()] or [*retry*(), *substitute*()].

6 Validation

The correctness of the repairability reasoner is defined in terms of its *completeness* and *soundness*. The WfRR is complete if every time the repair planner finds a repair plan for a faulty workflow instance the activities repaired by the repair plan are regarded as repairable by the repairability reasoner. The WfRR is sound if the repair planner is unable to generate a repair plan (for every workflow instance) in all the cases for which the WfRR returns an empty list of repairable activities. To evaluate these properties we decided to run the WfRR on a set of randomly generated workflows and compare the results with those obtained from running the repair planner on the same set.

The initial experimental setup consisted in running the tools on five different test cases. Each test case consists of a number of workflow schemas and each schema is generated with a random number of activities, variables (we show the results for cases with a maximum of 20 activities) and its associated repairability information, including failure and branching probabilities, goal variables, predefined repair handlers and compensating/equivalent activities. Additionally, for each schema we randomly generate the corresponding conversation models. Since we are only interested in faulty executions, for each schema we generate a random faulty instance which contains a random number of abnormal activities.

Table 1 shows the results of the experiments conducted on five different cases. For example, case 1 consists of 40 Wf schemas with 12 activities (2 of which are XOR-splits) and 4 abnormal activities. For this case, the repair planner found a repair plan for only 6 of the Wfs and, for each of these 6 Wfs, the list of repairable activities generated by the WfRR is a superset of the list of activities repaired by the repair plan (completeness). Additionally, the WfRR found that only 3 (out of 40) Wfs contain an empty list of repairable activities. Accordingly, for each of these 3 schemas the repair planner did not find a repair plan (soundness). Note that in the cases for which the repair planner finds no repair plan (e.g. 34 out of 40 in case 1) the WfRR may find that some activities are repairable. This is because the repairability ultimately depends on instance data.

Table 1. Evaluation Results

Case	Schemas	Act.	XOR	Abn Act.	Wf with Rplan	Complete	Non-rep-Wfs	Sound
1	40	12	2	4	6(15%)	6(100%)	3(7.5%)	3(100%)
2	50	20	4	2	21(42%)	21(100%)	0	N/A
3	100	10	2	4	19(19%)	19(100%)	6(6%)	6(100%)
4	150	12	2	4	17(11.3%)	17(100%)	7(4.67%)	7(100%)
5	1000	12	2	4	1000	1000(100%)	52(5.2%)	52(100%)

7 Conclusion

The main contribution of this paper is a heuristic-based algorithm for reasoning about repairability of Web service-based workflow processes at design time.

Such algorithm evaluates a workflow definition taking into account a number of repairability factors with the aim of identifying design flaws that lead to non-repairable activities. The results of the analysis are then fed back to designers in order to support the design of repairable workflows. The algorithm works on the workflow schema and assumes the designer is able to specify certain repairability information along with a model of the conversations supported by the workflow with each partner service. The algorithm weights non-repairable activities according to their relative importance w.r.t the repairability of the workflow.

As for future work, we would like to improve the current reasoning algorithm. One possibility is to incorporate the diagnosability aspect of workflows into the reasoning method in order to consider the diagnosability of activities. Although the initial evaluation results match our expectations further testing needs to be done in order to evaluate the correctness of the algorithm on a much bigger set of workflows and faulty instances.

References

1. W.M.P. van der Aalst, ‘The Application of Petri Nets to Workflow Management’, *J. of Circuits, Systems and Computers*, **8**(1), 21–66, (1998).
2. N. R. Adam, V. Atluri, and W.-K. Huang, ‘Modeling and analysis of workflows using petri nets’, *J. Intell. Inf. Syst.*, **10**(2), 131–158, (1998).
3. H. Ascher and H. Feingold, ‘Repairable systems reliability: Modeling, inference, misconceptions, and their causes’, *J. of the American Statistical Assoc.*, **82**(397), 363, 1987.
4. D5.1, ‘Characterization of Diagnosability and Repairability for self-healing Web Services’, Technical report, WS-DIAMOND European project, (June 2007).
5. Johann Eder and Walter Liebhart, ‘Workflow recovery’, in *Conference on Cooperative Information Systems*, pp. 124–134, (1996).
6. C. Hagen and G. Alonso, ‘Exception handling in workflow management systems’, *IEEE Trans. Softw. Eng.*, **26**(10), 943–958, (2000).
7. M. Reichert and P. Dadam, ‘ADEPT flex - supporting dynamic changes of workflows without losing control’, *J. of Intelligent Information Systems*, **10**(2), 93–129, (1998).
8. Nick Russell, Arthur, Wil M. P. van der Aalst, and Natalya Mulyar, ‘Workflow control-flow patterns: A revised view’, Technical report, BPMcenter.org, (2006).
9. WS-Diamond Team, ‘Ws-diamond: Web services diagnosability, monitoring and diagnosis’, in *18th International Workshop on Principles of Diagnosis (DX 2007)*, Nashville, TN, USA, (May 29-31 2007).
10. Moe T. Wynn, H.M.W. Verbeek, Wil M. P. van der Aalst, Arthur H.M. ter Hofstede, and David Edmond, ‘Business Process Verification - Finally a Reality!’, *Journal of Business Process Management. (to appear)*, (2007).
11. M. Reichert and S. Rinderle and U. Kreher and P. Dadam, ‘Adaptive Process Management with ADEPT2’, in *ICDE ’05*, pp. 1113–1114, Washington, DC, USA, (2005). IEEE Computer Society.
12. Meng Yu and Peng Liu and Wanyu Zang, ‘Multi-Version Attack Recovery for Workflow Systems’, in *ACSAC ’03*, pp. 142, Washington, DC, USA, (2003). IEEE Computer Society.
13. Hernâni Mourão and Pedro Antunes, ‘Workflow Recovery Framework for Exception Handling: Involving the User’, in *CRIWG ’03*, pp. 159-167, Autrans, France, (2003). Springer.