Repair of service-based processes - an application area for logic programming*

Gerhard Friedrich

Universitaet Klagenfurt
Universitaetsstrasse 65-67
9020 Klagenfurt, Austria
gerhard.friedrich@ifit.uni-klu.ac.at

Abstract. The handling of failing process executions has a long history in computer science. In this article we introduce methods for the diagnosis and repair of failed service-based processes and describe how logic programming can be successfully applied to problem solving. We formulate desirable properties like correctness, completeness, repair optimality and implementation/maintenance efficiency for this task and outline the problem using an introductory example. Finally, we sketch our solution based on model-based diagnosis, planning and logic programming which satisfies the desired qualities.

1 Introduction

Coinciding with the growth of the Web, web service composition has recently become a popular method for constructing software systems involving multiple partners. Such orchestrated web services can be seen as distributed processes where calls to procedures are realized by invoking the operations of individual web services. Business process execution languages [5] are employed to define the control structure and the exchange of information between various operations. However, dealing with service failures has become increasingly important because of the complexity involved in combining multiple organizations as service providers.

In particular, repair mechanisms apply actions to incorrectly and possibly partially executed processes to try to ensure that the goals of the process are still achieved. In cases where this strategy succeeds, the faulty process execution is *repaired*. From a practical point of view the following properties are desirable:

- Correctness: The execution of a sequence of repair actions will guarantee the achievement of the process goals. Note that in practice the achievement of process goals is conditionally dependent on correctness/fault assumptions.
  These correctness/fault assumptions are expressed using a *set* of diagnoses. A diagnosis is a *set* of assumptions over past and future executions of activities that classifies these executions either as correct (an activity behaves as intended) or faulty (an activity does not behave as intended). E.g. a faulty process execution can

* This research has been developed in the FET project WS-Diamond (http://wsdiamond.di.unito.it) IST-516933.

be repaired under the assumption that a set of likely diagnoses (i.e. highly unlikely diagnoses are ignored) includes the real diagnosis.
- Completeness: If a repair exists for a set of diagnoses, the repair system can compute the required repair actions.
- Repair optimality: In situations where several alternatives exist for repairing a faulty process execution, the cheapest repair is generated based on a given cost function.

Currently, the state-of-the-art repair technique is programming conventional process specific exception handling code. Obviously, correctness is a requirement; however, usually completeness and repair optimality are also desirable. The obvious drawbacks are:

- Implementation costs: The code for repairing process executions has to deal with a combinatorial explosion of contingencies when all likely fault situations and possible repairs are considered. E.g. if multiple faults are possible simultaneously.
- Unclear quality: Given an exception handling mechanism it is unclear for which failure situations correctness, completeness and repair optimality can be guaranteed.
- Maintenance costs: Unfortunately, small changes in the process can lead to substantial changes being required in the exception handling code.

Logic-based methods have the power to provide a general solution for the repair of service-based processes. Advances in answer set programming [4] allow us to combine model-based diagnosis and planning methods to implement process repair systems. In the following we will outline the general idea with an example and close by outlining the current solution and future work.

2 Example

Our example process is depicted in Figure 1 (a). A process is defined by activities (e.g. STARTFLOW, $a0, \ldots,$ GUARD, ENDFLOW) which are connected by the usual control activities (e.g. xor, $join$). Every process starts with the activity STARTFLOW and is completed with ENDFLOW. Activities can read variables and output values to variables ($o1, \ldots, o4$). The output variables of STARTFLOW are the inputs to the process. The input variables of ENDFLOW are the outputs of the process. These variables are called the goal variables of the process (e.g. $o4$). A process execution terminates correctly if correct values are assigned to the goal variables after the process execution is completed, i.e. all of the activities involved behave as intended.

Process executions are monitored by observing the values of various variables. Monitoring activities (e.g. GUARD) read some variables and can output either FAILURE or OK. Signaling FAILURE has the implication that the combination of variable values read by a guard is not allowed in a correct execution of the process.

In web service scenarios, activities can be associated with the operations of web services. These associations can change over time, for example if a web service provider is replaced. Furthermore, web services can be stateful, with the execution of operations

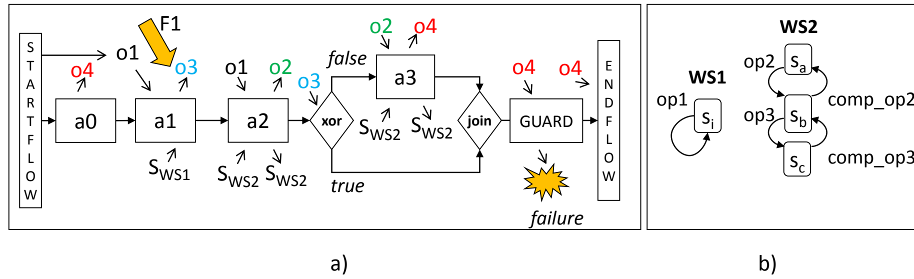

Fig. 1. A process (a) and a process execution with a failure and a fault F1, and design-time information (b) about the invoked services

changing the state of a web service. The state transition graphs of our example are depicted in Figure 1 (b). States of web services express pre-conditions for the execution of operations, e.g. $op3$ can only be executed if *WS2* is in state S_b.

In our example $a0$ is an internally implemented activity (e.g. a call of a stateless function implemented by the process owner). Activity $a1$ is linked to operation $op1$ of web service *WS1*, $a2$ to $op2$ of *WS2* and $a3$ to $op3$ of *WS2*. An activity linked to an operation of a stateful web service reads and writes the state of the web service. $a1$ reads the state of *WS1* denoted by S_{WS1}. $a2$ and $a3$ reads and writes the state of *WS2* denoted by S_{WS2}.

For practical reasons and because web services can be implemented by parties not under the control of the process owner, the formal semantics of the web service might be only partially known. E.g. only the interface, the state transition graph, and some informal input/output description is published. In addition, workflows activities may actually be realized by humans who follow an informal and cursory job description. Consequently, repair methods for faulty process executions often have to deal with an incomplete specification of the correct I/O behavior of activities.

In cases where a guard recognizes a failure, execution halts and the repair of the process execution starts. To facilitate repair a diagnosis step identifies the most *plausible* diagnoses, i.e. the most plausible worlds based on some preference relation. In the diagnosis field this preference relation is defined by probabilities or simply by the number of faults in the diagnoses. I.e. diagnoses with fewer faults are preferred over diagnoses with more faults. Requiring the correctness of repairs implies the achievement of the process goals in a set of possible worlds. Consequently, by reducing the set of *all possible* worlds to a set of *most plausible* diagnoses the number of repairs is increased, e.g. the effects of some highly unlikely multi fault diagnoses need not be considered in a repair. As a result, the repair costs are possibly reduced, or even repairs may exist for the most plausible diagnoses whereas there are no repairs for all possible worlds. Note, that the diagnosis of faulty process executions is not simply a matter of tracing dependencies [3] because the semantics of some activities are partially known (e.g. the xor control activity) and thus some fault assumptions can be inconsistent.

Typically, various repair actions are available for different activities. The most common repair actions include the re-execution of an activity, the substitution of a web service and compensating for an activity execution. Compensating for the execution of activity a_i will cause the variables affected by a_i to revert to the values they had right before the execution of a_i. Furthermore, compensation actions can require preconditions. In particular, compensations for the operations of a web service can only be executed if the web service is in a specific state. E.g. the compensation of $op2$ of *WS2* named $comp_op2$ can only be executed if *WS2* is in state S_b.

Referring to the process of Figure 1, let us assume that activities STARTFLOW, $a0$, $a1$, $a2$, xor, $a3$, JOIN and GUARD have been executed, and that the execution of GUARD throws an exception. In this example we focus on the repair task and therefore let us assume that a diagnosis process signals that the execution of $a1$ is faulty and that the faulty behavior is *transient*. In our example, the fault in $a1$ may have affected variable $o3$.

Consequently, the execution of the xor could lead to faulty branching. As a result, a correct execution of the process might not have required the execution of $a3$ and therefore the goal variable $o4$ could be in a faulty state. Consequently, the execution of ENDFLOW cannot guarantee the correctness of the process output.

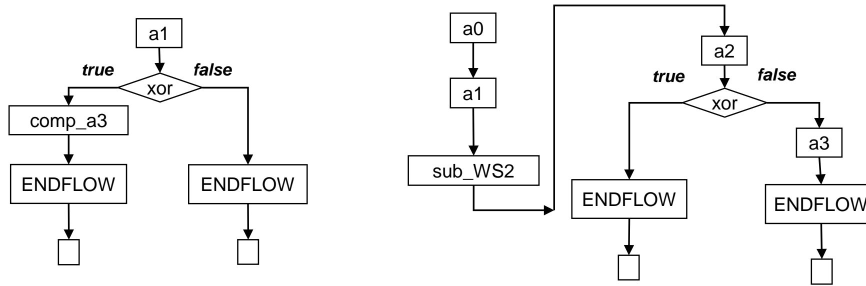

Fig. 2. Repair plans for example process instance

Let us initially assume for this example that operations $op2$ and $op3$ (and therefore also activities $a2$ and $a3$) can be compensated for and that all of the web services can be substituted. Our methods take the available repair actions as an input.

Using such an input, the left hand side of Figure 2 shows a repair plan which produces end states such that the value of goal variable $o4$ is a correct process output. A re-execution of $a1$ generates a correct state of $o3$. Note, that $a1$ can be re-executed with no compensations because $a1$ did not change the state of its service instance and the input $o1$ was not altered by the process execution. Next, the xor is re-executed. After re-executing $a1$, the exact output value of $a1$ is not known for $o3$ because of the unknown I/O behavior of $a1$. It is just known that $a1$ produced some value for $o3$. This unknown value serves as input to the execution of the xor. Subsequently, because the

execution of the xor reads some unknown value, the branching behavior is not known. Depending on the output of the xor execution, the repair plan branches: If the output is *true*, then the effect of $a3$ on $o4$ is compensated for, i.e. this yields a correct state of the goal value $o4$. In case the re-execution of xor evaluates to *false*, no further repair actions are needed as $o4$ is already in the correct state, because the output of $a3$ depends only on value $o2$ which is correct.

Let us assume that, for some reasons, a compensation of $a3$ is no longer available and thus the above repair plan cannot be applied. In this case, alternative repair plans are available, such as the one depicted on the right hand side of Figure 2. Re-executing $a0$ and $a1$ produces the respective correct states of $o4$ and $o3$. Next, the repair system substitutes web service *WS2* with an equivalent web service and the process is completed as defined in Figure 1.

By varying the availability of repair actions as well as the number, type (permanent vs. transient) and location of faults (i.e. the set of diagnoses), a range of repair plans can be generated. Clearly, designing repair handlers manually would be impractical as it would require the anticipation of all possible contingencies.

Note that the plan on the right-hand side of Figure 2 is valid also for the first scenario, where $a3$ can be compensated for. In fact, depending on the costs of executing repair actions (e.g. repair time), one of these repair plans depicted in Figure 2 is preferable. The proposed methods based on logic programming allow the generation of various repair plans for a given case, the best of which can be selected based on cost.

3 Current solution and future work

Our current solution includes a diagnosis system and a plan generator. After an exception is detected, a diagnosis step is performed and the most plausible diagnoses are passed to the plan generator which computes a repair plan.

For the diagnosis step [3] we exploit model-based techniques, where the behavior of activities is modeled using constraints. In particular, we had to develop a new diagnosis method for identifying incorrect activities in process executions because exact behavioral models for the activities are not available and dependency-based methods (which can deal with missing behavior descriptions of activities) accept fault assumptions although they are inconsistent. In [3] we formally characterize the diagnosis problem and develop a symbolic encoding that can be solved using CLP(FD) solvers. The current implementation is based on ECL^{i}PS^e Prolog (see `http://eclipseclp.org`).

Our solution for generating repair plans [2] is based on DLV [4] (see `http://www.dlvsystem.com`) and exploits the ideas of [1] on encoding planning problems by disjunctive logic programming. We designed the set of clauses such that every logical model contains exactly one contingency plan representing a repair plan. As presented in our example, repair plans correspond to contingency plans because the execution of actions can result in various successor states which can be discriminated via observations (e.g. observing the outcome of an xor). In particular, logical models have the property that the values of the goal variables are correct for all end states of the contingency plan. Furthermore, we use soft constraints to compute optimal repair plans while minimizing the expected costs.

The knowledge-base is structured as follows:

- The application specific component: A logical description specifying the structure of the process, the availability of repair actions, the linkage of activities to operations of web services, the available web services for potential replacements and the costs of repair actions.
- The description of the repair case: A description of the sequence of activity executions and, if available, the outcomes of these executions including information about the guard executions and a set of diagnoses.
- The description of the general repair problem: Logical descriptions classifying the correctness of the value of variables and specifying the preconditions and effects of actions.

Our repair/diagnosis approach has the following properties:

- Correctness: Provided that the models correctly describe the real world, e.g. preconditions and effects are adequately described and the real diagnosis is contained in the output of the diagnosis step, we can guarantee that the repair plan will be successful, i.e. correct.
- Completeness: Because we can characterize the set of all repair plans employing a logical description, we can guarantee completeness by employing complete reasoning systems. However, the computational intractability of the problem and general resource limitations mean that tractable completeness guarantees can be made only for subclasses of the general problem.
- Repair optimality: The full advantage of a complete description of repair plans is exploited for optimization. Local search methods can be applied to improve solutions within a time window where an unlimited expansion of the time window guarantees the optimality.
- Maintenance costs: The logic-based approach is also modular in the sense that the description of new activities or changes of the process structure only require changes within the application specific part and not of the general problem definition. The human effort involved in constructing exception handling is reduced by reusing the general description of the diagnosis/repair problem.

Our future work will expand the classes of problems where tractability can be guaranteed. Regarding knowledge representation, the observant reader will have noticed that methods for dealing with unknown values correspond to existential quantification. Currently, we have implemented existential quantification by introducing symbolic values as proposed in the area of program analysis. Logical languages which permit the safe and efficient application of existential quantification are desirable for diagnosis and repair because domain specific encodings and workarounds can be avoided.

References

1. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning: Semantics and complexity. ACM Trans. Comput. Log. 5(2), 206–263 (2004)

2. Friedrich, G., Fugini, M., Mussi, E., Pernici, B., Tagni, G.: Exception handling for repair in service-based processes. IEEE Trans. Software Eng. 36(2), 198–215 (2010)
3. Friedrich, G., Mayer, W., Stumptner, M.: Diagnosing process trajectories under partially known behavior. In: ECAI 2010 - Proceedings of the 19th European Conference on Artificial Intelligence. Frontiers in Artificial Intelligence and Applications, vol. 215, pp. 111–116. IOS Press (2010)
4. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Log. 7(3), 499–562 (2006)
5. Modafferi, S., Mussi, E., Pernici, B.: SH-BPEL: a self-healing plug-in for Ws-BPEL engines. In: MW4SOC '06 - Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006). pp. 48–53. ACM, New York, NY, USA (2006)