Diagnosis of Service Failures by Trace Analysis with Partial Knowledge

Wolfgang Mayer[1] and Gerhard Friedrich[2] and Markus Stumptner[1]

[1] Advanced Computing Research Centre, University of South Australia, [mayer|mst]@cs.unisa.edu.au
[2] Alpen-Adria Universität, Austria, gerhard.friedrich@uni-klu.ac.at

Abstract. The identification of the source of a fault ("diagnosis") of orchestrated Web service process executions is a task of growing importance, in particular in automated service composition scenarios. If executions fail because activities of the process do not behave as intended, repair mechanisms are envisioned that will try re-executing some activities to recover from the failure. We present a diagnosis method for identifying incorrect activities in service process executions. Our method is novel both in that it does not require exact behavioral models for the activities and that its accuracy improves upon dependency-based methods. Observations obtained from partial executions and re-executions of a process are exploited. We formally characterize the diagnosis problem and develop a symbolic encoding that can be solved using constraint solvers. Our evaluation demonstrates that the framework yields superior accuracy to classic dependency-based debugging methods on realistically-sized examples.

1 INTRODUCTION

The proliferation of orchestrated Web Services has increased the importance of diagnosing errors in process executions. Diagnosing the execution of concurrent applications is a challenging task even in local environments, but is exacerbated in large scale distributed business interactions, as acknowledged in a recent IEEE TSC special issue on Transactional behavior. Orchestrated Web Services define a process where individual activities are implemented by Web Services. If individual activities fail during execution, raising exceptions, repairs must be carried out [8], but as the authors explain, while languages such as WS-BPEL provide exception handling facilities, the construction of the logic to conduct fault handling is time consuming and itself error prone. (The other option at this point would be to fall back on compensation approach.) To provide complete and correct methods for repair, a complete and correct diagnosis method is of central importance. The goal of this paper is to infer minimal (irreducible) *diagnoses*, or problematic service behaviors that need to be avoided (e.g,. through re-execution) in terms of activity executions from observed execution traces.

While powerful techniques for runtime verification [5] or specification of fault handling logic [2, 8] have been proposed, essentially they presuppose the existence of a detailed specification of intended service behavior in one case, and detailed fault models in the other. Unfortunately, these assumptions are not necessarily generally satisfied in practice. The correct control flow may be specified but precise models of individual services and activity behaviors are usually unavailable. Fortunately, the sequence of activity executions can be obtained from the execution engine. However,

in case of failures (i.e., if exceptions are triggered), a repair-enabled execution engine needs the ability to execute and re-execute activities in order to achieve a successful process execution despite the fault. This increases the difficulty of the task, since repair executions (re-executions) do not necessarily follow the defined control flow. Our only assumption is that raising a fault will raise an exception.

To solve the problem of partial knowledge, earlier work has used dependency tracing [14, 1]. As we will show in our example, such methods cannot always correctly compute the set of minimal diagnoses because they do not fully capture the semantics of the employed control elements. Also, to the best of our knowledge no current generic diagnostic approach can deal with (re-)executions of activities, nor deal with partially known behaviors.

We present an approach to isolate minimal sets of faulty activity executions based on the structure of a given process while assuming that the behavioral descriptions of individual activities may not be given fully. Our approach relies on partial models of individual activities that are gathered from observed input and output values that occur in execution traces. No complete formal specification of an activity is required.

The paradigm of *consistency-based diagnosis* [13] is based on the assumption that faults are expressed via inconsistencies between *observations* (observed results of the actual system behavior) and the expected system behavior. In our case, such inconsistencies are the result of raising an exception. A *diagnosis* specifies the set of observed activity executions that are assumed to be correct. These assumed-correct activity behaviors must be part of "guaranteed safe" behavioral models for the activities of a process definition s.t. (i) no exceptions will be triggered for all possible process executions and (ii) specified activity behavior constraints are fulfilled. Such behavior constraints express *partial* knowledge about activity behaviors. If such a process behavior cannot exist, then some activity behaviors must be incorrect. The lack of precise knowledge about activity behaviors creates the necessity to reason about all possibly correct behaviors of activities. We tackle this problem by introducing sets of possible behavior descriptions and the propagation of symbolic constants representing specific but unknown values that may be created during execution of the process. Our approach is highly flexible; particular workflow patterns such as XOR splits (our example in this paper) are merely special cases of activities with particular observed behaviors.

We develop a correct and complete diagnosis method for a sequence of activity (re-)executions. We introduce basic concepts and an example in Sec. 2 and present the process model in Sec. 3. In Sec. 4 we provide the diagnosis concepts for process (re-)executions. Sec. 5 introduces the diagnosis method based on symbolic values. Its implementation and evaluation are discussed in Sec. 6.

2 EXAMPLE

We use the example depicted in Figure 1 to introduce core concepts of our approach. The upper part of this figure shows the process definition, the lower part depicts the executions of activities.

The process definition includes processing activities (e.g. SAMPLE) connected by a control-flow using XOR-splits (i.e. X1) and XOR-joins (i.e. J1 and J2) as control activities. Activities read input variables and store their results in output variables. Process executions are started by the execution of activity START which provides the process inputs. A process execution is finished by the execution of END. The outputs

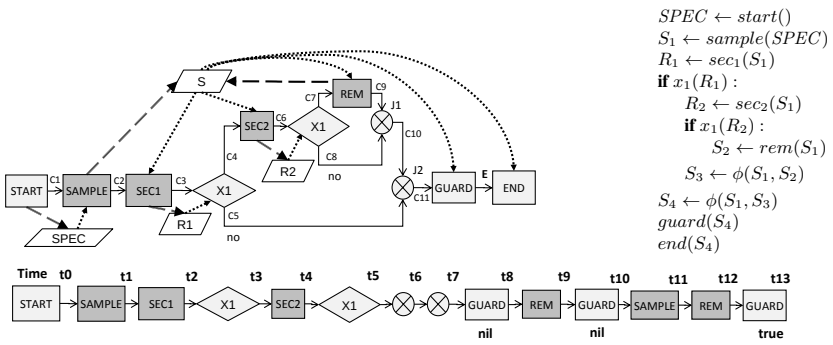

Fig. 1. Example process (top left), its Static Single Assignment form (right) and a sequence of activity executions (bottom)

of a process are the inputs of activity END. In our example the input to the process is a specification of a test sample (variable $SPEC$) which is used by activity SAMPLE to generate a sample placed at S. S is inspected by SEC1 and SEC2. Depending on the outcomes of SEC1 and SEC2, activity REM is eventually executed to remove some parts of the sample. Before ending the process a guard examines the sample for a final quality control. This guard can decide that the process failed by assigning nil to the control variable E thus stopping the execution. In keeping with other work in the area, e.g., [5], we assume for simplicity and without loss of generality, that a service has one operation and the invocation of that operation equates execution of the service.

Assume that the process was executed as shown in Fig. 1. Time points mark the end of an executed activity. The completion of activity executions are observed. GUARD raises an exception by assigning nil to E at time t8. We assume that only the processing activities SAMPLE, SEC1, SEC2, REM could be faulty. Given the flow of execution, activity executions SAMPLE_{t1} and SEC2_{t4} are the only ones that could have failed. $\langle \text{SAMPLE}_{t1}, \text{SEC2}_{t4} \rangle$ is the only minimal conflict so far; a correctness assumption of SEC1_{t2} is not needed to predict that the guard will fail. Both branches of the first occurrence of X1 in the process will lead to an execution of GUARD that fails if SAMPLE_{t1} and SEC2_{t4} are assumed to be correct. Diagnosis methods based on tracing dependencies [14] would *not* exonerate SEC1_{t2} since the computation of E depends on the output of SEC1_{t2}. (Recall that dependency based models implicate all activities that contribute to the derivation of an inconsistency in an execution trace. Our model is stronger in that we also consider inferences that involve hypothetical, unknown output values of activities. Hypothetical values allow us to prove that some explanations derived from a dependency based approach are in fact incorrect and result in an exception.)

Let us assume that a failure of SAMPLE_{t1} is unlikely, so $[\text{SEC2}_{t4}]$ is the only leading diagnosis. It follows, that SEC2 must output to R_2 a value such that the second occurrence of X1 takes the upper branch. REM has to be executed to avoid the exception.

Let us assume a repair reasoner decides to execute REM and GUARD after the execution of GUARD at t8, but the execution of GUARD_{t10} generates another exception. It follows that $\langle \text{SAMPLE}_{t1}, \text{REM}_{t9} \rangle$ is a further minimal conflict. Whatever branch is taken in the process, assuming executions SAMPLE_{t1}, REM_{t9} as correct leads to

an exception raised by GUARD. Consequently, [SAMPLE$_{t1}$] is the only single fault diagnosis. So the repair reasoner decides to execute SAMPLE again at t11. If we assume that the second execution of SAMPLE (at t11) outputs the same value as the execution of SAMPLE at t1, then the diagnosis [SAMPLE$_{t1}$] has to be extended to [SAMPLE$_{t1}$, SAMPLE$_{t11}$].

Consequently, there are two minimal diagnoses [SAMPLE$_{t1}$, SAMPLE$_{t11}$] and [SEC2$_{t4}$, REM$_{t9}$]. If the first diagnosis is very unlikely (since we know that the probability for SAMPLE to fail twice is an order of magnitude lower than the second diagnosis) then the repair reasoner decides to execute REM again which now provides a different value than REM$_{t9}$. Next GUARD is executed which returns t (true). At this point we can conclude that the value provided by REM$_{t12}$ corresponds to a value where the process is executed for the input provided by START$_{t0}$ and all activities worked correctly. Consequently, the faulty execution of the process is repaired.

In such a diagnosis/repair scenario two challenges must be addressed. (1) The execution follows the control path of the process definition until an exception is raised. At this point a repair reasoner takes over control and (re-)executes activities in an order that may differ from the one specified in the process definition. Note, if an activity can be re-executed is decided by the repair reasoner.

(2) It *cannot* be assumed that a complete definition of the behavior of the activities is available. In many cases only the structural description of the process and the execution trace is provided.

To deal with partially known behavior we present a process model that allows to define sets of possible activity behaviors. (For brevity, we will not address the mapping to and from BPEL and remain within the formal notation.)

3 PROCESS MODEL

In our model, a process consists of activities that are connected by shared variables. To obtain a model that is suitable for simulation and diagnosis, the semantics of each activity and the control and data flow between activities must be captured. We follow the proposal of [11] and represent the semantics of the process as constraints over the process variables. Different from previous models, our approach explicitly captures alternative possible process behaviors in a single model. Our notation is based on Reiter's logic formalism [13], but the underlying ideas apply to other formalisms, such as transition systems. We first describe the flow-related modeling aspects:

Definition 1 (Process). *A process $P = \langle A, V, I, O\rangle$ consists of a set of literals $A = \{A_1, \ldots, A_n\}$ representing activities. Occurrences of each activity are defined over a set of process variables V. $I \subset V$ and $O \subseteq V$ represent the input and output variables of P, respectively.*

Each occurrence of activity A_i in P receives a vector of input values through some process variables (the vector of variables is denoted by $\widetilde{I}_i$) and outputs a vector of values to process variables (denoted by $\widetilde{O}_j$). Activities A_i may occur several times in the process exploiting different process variables. A process has a distinguished START activity with no predecessors and an END activity with no successors. Processes conform to the Static Single Assignment (SSA) form [3], values through some process variables and outputs values to some process variables. The vector of process variables serving as input

(output) for A_{i_j} is denoted by $\widetilde{I}_{i_j}$ ($\widetilde{O}_{i_j}$). A process has a distinguished START activity with no predecessors and an END activity with no successors. Processes conform to the Static Single Assignment (SSA) form [3], where each variable is defined by exactly one activity. This is accomplished by creating new indexed "versions" of variables and by introducing so called ϕ-activities that are placed at control flow join points. The SSA form of our example process is shown (in pseudo-code syntax) in Figure 1.

The input variables taken by a process are defined by the START activity, and the output variables are inputs to the END activity. The structure of P is expressed as the conjunction of all its activity occurrences

$$P(V) = \bigwedge A_i(\widetilde{I}_j, \widetilde{O}_j), \qquad A_i \in A; \widetilde{I}_j, \widetilde{O}_j \subseteq V; i \in [1, n]$$

that defines the control and data flow admitted by the process. As noted, an activity A_i may occur several times in $P(V)$. We use upper case letters to denote variables in first-order logical sentences. We write $P(\widetilde{I}, \widetilde{X}, \widetilde{O})$ to denote the conjunction $P(V)$ where the process input variables are bound to input values $\widetilde{I}$, the output variables are bound to $\widetilde{O}$, and the remaining process-internal variables are bound to $\widetilde{X}$. Predicates A_i govern the *allowed* value combinations admitted by the correct behavior of all occurrences of activity A_i. Hence, value assignments to all process variables $\widetilde{I}, \widetilde{X}, \widetilde{O}$ which satisfy the predicates of the activities A_i in the conjunction $P(\widetilde{I}, \widetilde{X}, \widetilde{O})$ correspond to the allowed execution(s) where P receives input values $\widetilde{I}$ and produces output values $\widetilde{O}$. A value assignment that satisfies all predicates A_i in $P(\widetilde{I}, \widetilde{X}, \widetilde{O})$ is an *execution* of the process. For simplicity of presentation, we assume that END has only a single control input variable E that indicates success or failure of a process execution. The SSA from of the example process is represented as the conjunction

$$\begin{aligned} P(SPEC, R_1, R_2, S_1, \ldots, S_4, C_1, C_2, \ldots, E) = \\ start(C_1, SPEC) \wedge sample(C_1, SPEC, C_2, S_1) \wedge \\ sec1(C_2, S_1, C_3, R_1) \wedge x1(C_3, R_1, C_4, C_5) \wedge \\ sec2(C_4, S_1, C_6, R_2) \wedge x1(C_6, R_2, C_7, C_8) \wedge \\ rem(C_7, S_1, C_9, S_2) \wedge \phi(C_8, C_9, C_{10}) \wedge \phi(S_1, S_2, S_3) \wedge \\ \phi(C_5, C_{10}, C_{11}) \wedge \phi(S_1, S_3, S_4) \wedge guard(C_{11}, S_4, E) \wedge end(E, S_4) \end{aligned}$$

where the variables C_i and E model the control flow and the remaining variables model the data flow. Control- and data flow joins are uniformly represented as ϕ-activities.

From here on we define the relation describing the behavior of an activity over a set of *activity variables*. We focus on the possible relationships between input and output values of an activity and do not rely on detailed knowledge about the internal structure or implementation of an activity. Since an activity may occur several times in P, the activity variables ($\bar{\cdot}$) may be bound to different process variables ($\check{\cdot}$) as shown in the example for X1. That is, the activity variables in the definition of the behavior relation serve as a placeholder for process variables.

Definition 2 (Behavior Relation). *Let A be an activity with activity variables $U_1, \ldots, U_t$ where the input variables are $\overline{I} = \langle U_1, \ldots, U_s \rangle$ and the output variables are $\overline{O} = \langle U_{s+1}, \ldots, U_t \rangle$, and let D_{U_k} denote the value domain of variable U_k. The allowed behavior of activity A is given as a relation over the allowed input and output values: $A(\overline{I}, \overline{O}) \subseteq D_{U_1} \times \cdots \times D_{U_s} \times D_{U_{s+1}} \times \cdots \times D_{U_t}$.*

We require that A is total, that is, $A(\widehat{v}, \overline{O})$ includes at least one tuple for each $\widehat{v} \in D_{U_1} \times \cdots \times D_{U_s}$. We describe the behavior relation of A extensionally by a set of literals.

Value domains correspond to types and can appear in multiple behavior descriptions. For example, the domain of the data output of SAMPLE is the domain of the processing input of SEC1. We require processes to be well typed such that an activity is defined on all values that could be produced by its predecessors. Without loss of generality we assume that any two domains are either equal or mutually disjoint.

Definition 3 (Process Behavior). *A process behavior B_P for a process P is a vector of activity behavior relations $\langle A_1(\overline{I}_1, \overline{O}_1), \ldots, A_n(\overline{I}_n, \overline{O}_n)\rangle$. $\overline{I}_i, \overline{O}_i$ denote vectors of activity variables.*

To accurately model the flow of control in a process execution, we assume that each domain D_{U_k} contains a distinguished symbol nil that represents "no value" and that is different from any value produced by any execution of an activity. The control flow between activities is expressed as a shared variable connecting each predecessor activity to its successor(s). Control activities AND-split, AND-join, and XOR-join are defined as usual where control input and output variables have the binary domain $\{t, nil\}$. For processing activities (those which process inputs and pass the control flow), guards, and XOR-splits, we amend the relation $A(\overline{I}, \overline{O})$ to include all tuples $\langle \widehat{v}, nil, \ldots, nil\rangle$ where an input value in $\widehat{v}$ is nil and all other input variables are bound to values of their domain. For ϕ-activities the output is nil iff both inputs are nil. We refer to these sets of tuples as the *nil-description*. This model derived from SSA form ensures that an activity produces non-*nil* outputs only if it is activated with non-*nil* inputs along the control flow path and produces nil otherwise. As a result, the control and data flow in any process execution are captured correctly. Furthermore, the model ensures that the END activity receives a non-nil control input iff the process runs to completion and does not raise an exception.

Let us now investigate the case where the behavior of an activity A is partially unknown. This situation may arise if we must predict the execution of a process on partial input or in the presence of fault assumptions. For example, the outputs of X1 cannot be predicted precisely without knowing the values supplied by SEC1 and SEC2. However, even if the behavior of SEC1 is not known, it is still possible to conclude that any execution of SEC1 will result in an assignment for R_1 and the activation of X1. Let the hypothetical value of the assignment to R_1 be r. Then it is known that X1 will activate either the upper or the lower branch. Consequently, the behavior relation of X1 will contain either $x1(t, r, t, nil)$ or $x1(t, r, nil, t)$ where the behavior of an XOR-split activity is expressed by the relation $x1(C, W, Y, N)$ defined over control variables C, Y, N and decision input W. Since XOR-splits exhibit deterministic behavior (for given inputs) the behavior relation could not contain both tuples. To capture this form of incomplete knowledge, a model must be able to express a set of possible behavior relations where each relation reflects a different possible behavior if complete information was available.

We generalize our model of an activity from a single relation to a set of relations in order to model the behaviors that may arise if the behavior relation is not known completely. The possible behaviors of an activity A are expressed by a set of relations $\mathcal{A}(\overline{I}, \overline{O}) = \{A^1(\overline{I}, \overline{O}), \ldots, A^z(\overline{I}, \overline{O})\}$, where each $A^k(\overline{I}, \overline{O})$ represents a behavior relation as defined previously.

E.g., the two possible behaviors of an XOR-split activity $x1(C, W, Y, N)$ with its decision input fixed to $W = x$ (where x may be nil) are

$$\{x1(nil, x, nil, nil), x1(t, nil, nil, nil), x1(t, x, t, nil)\} \text{ and}$$
$$\{x1(nil, x, nil, nil), x1(t, nil, nil, nil), x1(t, x, nil, t)\}.$$

More generally, if the value of variable W is not known, $\mathcal{A}_{X1}(\overline{I}_{X1}, \overline{O}_{X1})$ comprises all sets

$$\{\{x1(t, nil, nil, nil) \cup \bigcup_{x \in D_W} \{x1(nil, x, nil, nil), x1(t, x, Y, N)\}\} \mid \langle Y, N\rangle = \langle t, nil\rangle \text{ or } \langle Y, N\rangle = \langle nil, t\rangle\}$$

The behavior of the entire process P is determined as a combination of specific behaviors, one each from $\mathcal{A}_i(\overline{I}_i, \overline{O}_i)$ for all activities A_i in P. By constructing the set of possible selections we define the set of all possible process behaviors.

Definition 4 (Possible Process Behaviors). *The set of all possible behaviors of P is given as*

$$\mathcal{B}_P = \left\{ \left\langle A_1^{k_1}, \ldots, A_n^{k_n} \right\rangle \mid A_i^{k_i} \in \mathcal{A}_i(\overline{I}_i, \overline{O}_i) \right\}.$$

An element $B_P \in \mathcal{B}_P$ is a possible process behavior.

Assume an execution of P results in the following observed execution behavior of activities Obs:

$$\{start(t, spec_1), sample(t, spec_1, t, s_1), sec1(t, s_1, t, r1_1), x1(t, r1_1, t, nil), sec2(t, s_1, t, r2_1), x1(t, r2_1, nil, t), \ldots, guard(t, s_1, nil), rem(t, s_1, t, s_2), guard(t, s_2, nil), sample(t, spec_1, t, s_1), rem(t, s_1, t, s_3), guard(t, s_3, t)\}.$$

The same I/Os are observed for the executions of SAMPLE, while REM produces different outputs for the same input.

In absence of further information, the observed execution behaviors in Obs together with the nil-description comprise the behavior relations. Behavior relations of ϕ-activities and END are also included. Assume that REM may behave non-deterministically for some inputs, and that for the input value $r2_1$ the behavior of the XOR is unknown; that is, *no* behavior matching $x1(t, r2_1, \_, \_)$ has been observed. Then there are two possible process behaviors B_P^U and B_P^L for P: in B_P^U, the second occurrence of X1 in P activates the upper branch on input $r2_1$, while in B_P^L the lower branch is taken.

A given process behavior $B_P \in \mathcal{B}_P$ determines the set of possible executions of P. We abstract from the concrete execution(s) implied by a given B_P and project the process behavior on its output values:

Definition 5 (Reachable assignment). *Let B_P be a behavior of a process $P = \langle A, V, I, O\rangle$. An assignment of value w to output variable $\widetilde{Q} \in O$ is* reachable *under B_P iff some execution admitted by $P(\widetilde{I}, \widetilde{X}, \widetilde{O})$ satisfies $\widetilde{Q} = w$. We write*

$B_P \models \exists \widetilde{I}\, \widetilde{X}\, \widetilde{O} : P(\widetilde{I}, \widetilde{X}, \widetilde{O}) \wedge \widetilde{Q} = w$.

For the scenario described above it holds that in both possible process behaviors (B_P^U and B_P^L) $E = nil$ is a reachable assignment: $B_P^L \models P(SPEC, R_1, R_2, S_1, \ldots, S_4, C_1, C_2, \ldots, E) \wedge E = nil$ (the variables of P are existentially quantified). That is because the guard signals an exception both for s_1 and s_2. Assignments $S_4 = s_2$ and $S_4 = s_3$ are both reachable in B_P^U.

If B_P^U determines the execution, $E = nil$, because the guard signals an exception if $S_4 = s_2$ is reached. If B_P^U is changed to $B_P'^U$ by removing $rem(t, s_1, t, s_2)$ from the behavior relation of REM, $E = nil$ is no longer reachable in $B_P'^U$ but is still reachable

in B_P^L. The process behavior $B_P'^U$ specifies a process where –regardless of the concrete execution– no exception will be raised, whereas B_P^L admits an execution that fails. Consequently, if we assume that SEC2$_{t4}$ produces a different value than the observed value $r2_1$ and on this value the upper path of the second occurrence of X1 is taken, and REM produces a different value than s_1 or s_2 then we are guaranteed a process behavior which rules out exceptions.

4 DIAGNOSIS MODEL

In "black box" application domains such as Web Services the complete behavior relation $A_i(\overline{I}_i, \overline{O}_i)$ is unknown. However, we can exploit the available knowledge which on one hand specifies the I/O tuples that *must* be contained in a behavior relation and on the other hand describes which I/O tuples are *forbidden*. For each activity A_i a behavior relation $A_i(\overline{I}_i, \overline{O}_i)$ is defined to include the nil-description, all observations gathered from executions of activities, and possibly other known concrete I/O behaviors. This set of predefined behaviors is denoted by Pre. Some domains of the variables of A_i may be known, e.g., control variables, while others are only partially known (e.g. the output of $SAMPLE$). If we observe a value v of a variable O whose domain is only partially known, and v is not contained in this domain, we extend the domain with a new symbol representing v.

Additional requirements constraints Re_i determine whether the behavior of an activity A_i must be deterministic. Re_i is a constraint expression over the variables in $A_i(\overline{I}_i, \overline{O}_i)$ specifying which value combinations for activity variables $\overline{I}_i, \overline{O}_i$ are allowed in the behavior relation. For our purposes it is sufficient if Re_i refers to known domain values of $\overline{I}_i, \overline{O}_i$. These constraints are local to an activity and do not depend on the behavior of any other activity. The set of requirements for all activities is denoted by Re. The requirement that activity behaviors must be totally defined is part of Re_i.

Definition 6 (Diagnosis Problem). *A diagnosis problem $DP = \langle P, Obs, Pre, Re\rangle$ consists of a process P, a set of I/O behaviors Obs observed from executions, a set of predefined behaviors Pre, and a set of requirements Re. Let $P = \langle A, V, \emptyset, \{E\}\rangle$ with activities $A = \{A_1, \ldots, A_n\}$. Let $Obs = \{ob_1(\widehat{v}_1, \widehat{w}_1), \ldots, ob_q(\widehat{v}_q, \widehat{w}_q)\}$ be the set of observed I/O behaviors of activity executions, where $ob_j(\widehat{v}_j, \widehat{w}_j) \in Obs$ is the observed execution of an activity occurrence A_{i_j}. The set of all observed execution behaviors of an activity A_i is denoted by ob_i. $ob_i \subseteq A_i(\overline{I}_i, \overline{O}_i), A_i(\overline{I}_i, \overline{O}_i) \in Pre$ for $i \in \{1, \ldots, n\}$. Process variable E indicates success or failure of any execution of P.*

Note that without loss of generality, the definition limits P to a single output and does not mention process inputs: the inputs that were observed in executions are modeled as outputs of the START activity. Furthermore, the decisions that establish if a process execution is successful (typically referred to as an "oracle") are explicitly encoded in the guard activities of the process. Note that we do *not* require that the criteria are completely known and formalized. Rather, the behavior of guard activities is also determined by observations in Obs. Our model implies that if a guard activity determines that its input values violate a process constraint, a vector containing nil values will be assigned to its output variables. By the definition of the SSA form and the behavioral relations, nil will be propagated to the END activity by the subsequent activities. Hence, it is sufficient to verify that the END activity does not receive a nil value to verify that the process execution complies with all guards.

For example, the behavior Pre_{X1} of $x1(C, W, Y, N)$ is given by the set $\{x1(t, nil, nil, nil), x1(nil, X, nil, nil) \mid X \in D_W\} \cup Obs$ where Obs contains the observations $\{x1(t, r1_1, t, nil), x1(t, r2_1, nil, t)\}$. The requirements Re_{X1} for the XOR behavior are given by the following sentence:

$$x1(C, W, Y, N) \Rightarrow \\ [(C = nil \vee W = nil) \Leftrightarrow Y = nil \wedge N = nil] \vee \\ [(C \neq nil \wedge W \neq nil) \Leftrightarrow Y \neq N]$$

In addition Re_{X1} includes the property that $X1$ must be deterministic and totally defined.

If a failure occurred (indicated by a guard raising an exception) either during process execution or repair planner guided re-execution, some activity executions must have produced incorrect values. In other words, specific activity behaviors in the process are faulty, and the behavior definition must be restricted so that the incorrect I/O behaviors cannot occur. Conversely, behaviors do not need to be removed if their execution cannot result in a failure.

Let B_P be a process behavior. For a set of tuples Δ, $B_P \setminus \Delta$ is the process behavior where from each behavior relation in B_P the tuples of Δ are removed.

Definition 7 (Diagnosis). *Let $DP = \langle P, Obs, Pre, Re\rangle$ be a diagnosis problem with $P = \langle A, V, \emptyset, \{E\}\rangle$. A subset $\Delta \subseteq Obs$ of activity executions is a diagnosis for DP iff there exists a process behavior B_P such that*

1. *Each $A_i'(\overline{I}_i, \overline{O}_i) \in B_P$ is a superset of $A_i(\overline{I}_i, \overline{O}_i) \setminus \Delta$ for $A_i(\overline{I}_i, \overline{O}_i) \in Pre$*
2. *Each $A_i'(\overline{I}_i, \overline{O}_i) \in B_P$ is consistent with $Re_i \in Re$*
3. *$B_P \not\models \exists \widetilde{X}\, E : P(\emptyset, \widetilde{X}, E) \wedge E = nil$.*

Δ is minimal *if no $\Delta' \subset \Delta$ is a diagnosis for DP.*

The first condition expresses the key concern that the executions should be consistent with existing non-faulty activity behaviors, but omit the faulty behavior tuples. The second condition formalizes the expectation that activity executions must also satisfy general known requirements like totally defined. The third is the error-freeness condition for the diagnosed and repaired execution.

Hence, a diagnosis Δ rules out certain observed behaviors of activities, such that *no process execution* conforming to the remaining assumed-correct behavior relations in B_P can lead to a failure. We say Δ is accepted as a diagnosis iff there exists a correct process behavior B_P that extends $Pre \setminus \Delta$. A minimal diagnosis preserves as much as possible the observed behavior. If the same behavior of an activity is observed multiple times in an execution (e.g. $sample(t, spec_1, t, s_1)$) then either all of these executions must be correct or all must be faulty. This assumption introduces dependencies between activity executions and may affect the diagnosis probability. Devising suitable probability models is beyond the scope of this paper.

As examples, consider the following diagnosis scenarios assuming that the process was executed until the first execution of the guard returns a failure. All the behavior relations of the activities contain just the observed I/O behaviors and the nil-description. Re contains the usual restrictions on the allowed behavior of processing activities and control activities. If $\Delta = \emptyset$ then $E = nil$ is reachable, so $\Delta = \emptyset$ is not a diagnosis. If $\Delta = \{sample(t, spec_1, t, s_1)\}$ then we can construct behavior relations for all activities such that $Pre \setminus \Delta$ is extended and the process behavior is correct. For example, the execution of SAMPLE generates a new value for which we can assume that the guard does not signal a failure. However, if $\Delta = \{sec1(t, s_1, t, r1_1)\}$ then it is not possible

to generate a correct process behavior by extending $Pre \setminus \Delta$. Whatever value SEC1 generates, either the upper branch or the lower branch of the first occurrence of X1 in P is taken. In both cases, s_1 will be assigned to S_4 and therefore the guard will output nil (as in the original execution). Thus, $\Delta = \{sec1(t, s_1, t, r1_1)\}$ is not a diagnosis, as it does not prevent the exception.

In the following presentation we will assume that P is acyclic. This does not limit the representation of observed execution traces (traces are usually represented as partially ordered set of activities). But loops must be taken into account when projecting unseen behavior forward through the process, using common techniques to determine a sufficient number of unfoldings that cover all possible looping behaviors [7].

5 SYMBOLIC REPRESENTATION

To verify if Δ is a diagnosis, behavior relations $A'_i(\overline{I}_i, \overline{O}_i)$ of activities $A_i \in A$ must be found that include the tuples of $A_i(\overline{I}_i, \overline{O}_i) \setminus \Delta$, are consistent with Re_i, and no guard fails, i.e. $E = nil$ cannot be reached. If no such set of behavior relations exist then Δ is not a diagnosis. Consequently, all possible behavior relations of A_i have to be explored. If all domains of I/O variables of A_i are known we can enumerate all behaviors which are superset of $A_i(\overline{I}_i, \overline{O}_i) \setminus \Delta$ and consistent with Re_i. However, if domains are only partially known then we have to deal with unknown values.

We adopt the principle of *symbolic execution* [6] from program analysis to deal with unknown behaviors. In symbolic execution, unknown values of input and output variables of program statements are represented as symbols. Every occurrence of an activity A_i in the process P may produce a new, yet unseen value for a variable whose domain is partially unknown.

For an activity A_i and an output variable O of this activity, we inject unique symbols $s_1, \ldots, s_p$ into the domain D_O, where p is the number of occurrences of A_i in P. The domain D_O may be used multiple times by the same activity but also by other activities as a domain for output variables.

For example, assume that activities SEC1 and SEC2 use the same domain D for their data output. From observations we know that $\{r1_1, r2_1\} \subset D$. Both SEC1 and SEC2 can produce symbolic values $y1$ and $y2$ that represent yet unseen values in D. Since the symbolic values are not constrained further, both activities may output an arbitrarily chosen value —the same value or different values— in D. Hence the symbolic behavior relation must consider the cases where both activity executions result in the same symbolic value and where the values differ.

In the following we construct every possible behavior of activities given a diagnosis problem and a $\Delta \subseteq Obs$.

Let A be an activity with input variables $\overline{I} = \langle U_1, \ldots, U_s \rangle$ output variables $\overline{O} = \langle U_{s+1}, \ldots, U_t \rangle$, and let D_{U_k} denote the domain of variable U_k. The set of all input vectors of an activity is $w_{\overline{I}} = \{\langle w_1, \ldots, w_s \rangle \,|w_1 \in D_{U_1}, \ldots, w_s \in D_{U_s}\}$. Likewise the set of all output vectors of an activity is $w_{\overline{O}} = \{\langle w_{s+1}, \ldots, w_t \rangle \,|w_{s+1} \in D_{U_{s+1}}, \ldots, w_t \in D_{U_t}\}$.

Based on the I/O vectors we can construct all possible behavior relations of activities. However, in such a relation, for each input vector, at most p output vectors need to be defined, since the activity can only occur p times in P; on each occurrence a different output vector can be returned. If an activity is deterministic then just one output vector is created for each possible input vector. Consequently, the set of possible behaviors

for an activity A_i is defined by behavior relations where for each input vector p output behaviors are chosen. The same output vector may be selected multiple times.

$\mathcal{A}'_i(\overline{I}_i, \overline{O}_i) = \{\bigcup_{w_{\overline{I}} \in w_{\overline{I}}} \{\langle w_{\overline{I}}, w_1\rangle, \ldots, \langle w_{\overline{I}}, w_p\rangle\} | w_1 \in w_{\overline{O}}, \ldots, w_p \in w_{\overline{O}}\}$

All the possible behavior descriptions in $\mathcal{A}'_i$ are extended by the set of tuples considered to be correct, i.e. $A_i(\overline{I}_i, \overline{O}_i) \setminus \Delta$ for $A_i(\overline{I}_i, \overline{O}_i) \in Pre$, and by the nil-description. In addition, we eliminate all behavior descriptions in $\mathcal{A}'_i$ that are inconsistent with requirements Re_i. We generate the possible behavior for all activities which have variables with a partially unknown domain, such as processing activities, guards, XOR-splits, and ϕ-activities. The result is a set of possible process behaviors $\mathcal{B}_P$ for a diagnosis problem and a $\Delta \subseteq Obs$.

For example, the domain D_W of $x1(C, W, Y, N)$ is extended to contain all of $\{r1_1, r2_1, y1, y2\}$ (and nil as the only other value). Every possible behavior of X1 includes the tuples in Pre_{X1} (shown earlier) and must be consistent with Re_{X1}. That is, the behavior on inputs $\{r1_1, r2_1, nil\}$ is fixed, but there are four different behaviors which differ just on the outputs provided for inputs $\{y1, y2\}$:

$\{x1(t, y1, t, nil), x1(t, y2, t, nil)\}, \{x1(t, y1, nil, t), x1(t, y2, t, nil)\},$
$\{x1(t, y1, t, nil), x1(t, y2, nil, t)\}, \{x1(t, y1, nil, t), x1(t, y2, nil, t)\}.$

Based on the possible process behaviors $\mathcal{B}_P$ for a diagnosis problem DP and a diagnosis Δ we can state the following property which is exploited for the generation of diagnoses:

Property 1. Let $DP = \langle P, Obs, Pre, Re\rangle$ be a diagnosis problem with $P = \langle A, V, \emptyset, \{E\}\rangle$ and $\mathcal{B}_P$ the set of possible process behaviors generated for a subset $\Delta \subseteq Obs$ as described above.

Δ is a diagnosis for diagnosis problem DP iff there is a process behavior $B_p \in \mathcal{B}_P$ s.t. $B_p \not\models \exists \widetilde{X}\ E : P(\emptyset, \widetilde{X}, E) \wedge E = nil$.

Proof sketch: ($\Leftarrow$) : This is trivially fulfilled by the construction of $\mathcal{B}_P$. All activity behaviors in $\mathcal{B}_P$ are supersets of $A_i(\overline{I}_i, \overline{O}_i) \setminus \Delta$, are consistent with Re_i, and B_p does not trigger an exception.

($\Rightarrow$) : If there exists a diagnosis Δ for DP then there exists a process behavior B_P s.t. $B_P \not\models \exists \widetilde{X}\ E : P(\emptyset, \widetilde{X}, E) \wedge E = nil$. An instantiation of the variables in $B_P \models \exists \widetilde{X}\ E : P(\emptyset, \widetilde{X}, E)$ corresponds to a process execution and defines behavior tuples for activities A_i. Values not covered by observations are replaced by a symbol. By construction, at least one symbolic value is available for each occurrence of A_i. The introduction of symbolic values cannot trigger an exception, and all constraints in Re_i remain satisfied, since both cannot contain symbolic values. Thus, if a constraint is fulfilled for an arbitrary unknown value it is also fulfilled for a symbolic value. A_i remains to be totally defined after the substitution. The tuples of all process executions where the unknown values are replaced by symbolic values define behavior relations which are included in some behavior relation generated by our construction of $\mathcal{B}_P$. It follows that if there is a process behavior B_P for which $Pre \setminus \Delta$ can be correctly extended, then there exists a process behavior B'_P in the set of generated possible process behaviors $\mathcal{B}_P$ which is also a correct extension of $Pre \setminus \Delta$. We have constructed a decision method which determines if a set $\Delta \subseteq Obs$ is a diagnosis. □

Given our example process, observations and the diagnosis candidate $\Delta = \emptyset$, all $A_i(\overline{I}_i, \overline{O}_i)$ in the process behaviors of the generated set $\mathcal{B}_P$ contain their observed execution behavior. Therefore, in each process behavior of $\mathcal{B}_P$, $E = nil$ is reachable.

Consequently, there is no B'_P which is a correct extension of $Pre \setminus \Delta$. Hence, $\Delta = \emptyset$ is not a diagnosis. However, if the observed execution behavior $sample(t, spec_1, t, s_1)$ is removed, then only symbolic (unseen) output values remain to be assigned to S_1. Therefore, we can construct behaviors for REM, $GUARD$, and all other activities such that the guard is not triggered for any execution. Therefore, $\{sample(t, spec_1, t, s_1)\}$ is indeed a valid diagnosis.

6 Diagnosis Computation and Evaluation

Because all domains of the variables are finite, logic programming systems and model checkers can be used to concisely express all possible process behaviors and check whether $E = nil$ is reachable. The search for minimal, irreducible, or leading diagnoses can be implemented by standard methods, such as HS-tree generation, combined with appropriate minimization procedures, such as QuickXplain.

We conducted an empirical evaluation to determine the diagnostic accuracy that can be expected from our model, compare it with previous approaches, and assess the computational resources required. We sourced process examples from the literature, such as [4], and generated additional (artificial) processes to obtain a comprehensive benchmark suite of 200 processes. Each process comprised 5–79 activities chosen from 3–9 different types of activity, and each process included up to 22 xor decision nodes. Activities were assembled into complex processes based on a randomized graph grammar to ensure process control and data flow are well-defined. For each activity type, a set of observed behaviors was generated randomly to yield the observed process behaviors and exceptions. Two execution paths were generated for each process. The number of activities occurring in an execution path varied from 5–65 activities. The resulting benchmark suite of 400 process executions covers a wide range of different process structures, and, to the best of our knowledge, is more comprehensive than any other available benchmark.

We implemented the diagnosis framework in Eclipse Prolog. Each process was compiled into a finite domain constraint satisfaction problem which captured the structural and behavioral links between the activities. Concrete and symbolic values were encoded as integers to leverage efficient constraint solvers. We used additional variables to model correctness assumptions and the selection of possible behaviors. The constraint system was then used to isolate the maximal subsets of the observed process behaviors that did not result in an exception.

Size	N	Xor	Trace	Dep.	Symb.	Imp.	Time (s)
0–9	48	0.67	8.04	3.98	3.50	**0.10**	0.12
10–19	100	3.10	16.84	6.40	4.63	**0.16**	1.07
20–29	124	5.21	27.16	8.97	4.89	**0.45**	5.43
30–39	60	7.67	33.43	9.67	5.37	**0.59**	16.85
40–49	28	10.21	41.64	12.36	6.31	**0.50**	74.52
50–59	12	12.17	35.17	10.42	6.29	**0.32**	124.22
60–69	16	14.62	30.75	9.12	4.50	**0.40**	130.63
70–79	6	17.00	35.67	9.33	2.00		255.80
0–79	400	5.85	24.95	8.19	4.82	**0.29**	14.56

Table 1. Comparison of dependency- and symbolic diagnosis model

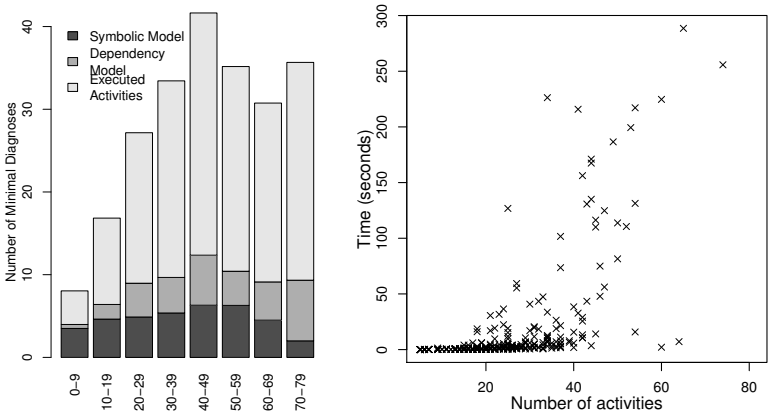

Fig. 2. Process size vs. result size and diagnosis time

Our results are summarized in Table 1, aggregated by process size. The columns show the average number of activities in a process (*Size*), the number of process executions considered in our study (*N*), the average number of decision nodes in a process (*Xor*), the average number activities in each execution trace (*Trace*), the number of minimal diagnoses obtained from dependency-based models (*Dep*) and from our model (*Symb*), the mean relative improvement (*Imp*$= 1 - Symb/Dep$), and the average diagnosis time in seconds (*Time*) to compute all diagnoses using the symbolic model. Accuracy is measured as the fraction of activity behaviors that need *not* be examined given a set of diagnoses. Note that *Imp* may be larger than $1 - Symb/Dep$ since some instances exceeded the five minute time limit for the symbolic model. The blank cell denotes "no improvement" and is caused by too few symbolic results. Hence, *Imp* is a conservative estimate and may improve further with faster algorithms.

The results show that the symbolic model yields significantly more accurate results than simpler dependency-based models. The symbolic model on average eliminated three diagnoses, but could shorten the result by as much as 20 diagnoses. Overall, the number of diagnoses dropped by roughly 30% compared to dependency-based models. Our model on average implicated only 21% of the executed activities. Figure 2 shows a bar plot of the additional spurious diagnoses that are incurred when moving from a more precise diagnosis model (lower bars in the diagram) to a more abstract model (mid- and upper bars). The greatest reduction of the diagnosis ratio was observed for process executions that contained a large number of activities. Among all diagnoses, 90% were single-fault explanations, 9.5% double-faults, and 0.5% triple-faults.

The measured execution times indicate that the symbolic model also performed well in those scenarios that are most relevant for practical application. Figure 2 shows a scatter plot of the diagnosis times. In 75% of all cases, the result was obtained after just 5.3 seconds. On average, all minimal diagnoses were obtained after 14.56 seconds of CPU time.[1] Our results confirm that the model is sufficient to address the majority of practical process diagnosis scenarios, where the number of activities is virtually always

[1] The data were obtained from Eclipse 6.1 on Intel P4@1.86GHz with 6Gb RAM running Linux 2.6.

less than 50. (Larger scenarios are usually decomposed hierarchically, where the number of activities on each level is small. Our model is particularly suited for hierarchical diagnosis, since no detailed specification of the abstract activities' behavior is required.) We believe that further optimization of our naïve implementation will improve these results.

Another area for future work lies in the more detailed incorporation of particular service properties. A significant paradigm adopted by most papers on Web services is the CRP (compensatable, retriable, and pivot) model originally developed for Multi-database systems [10]. Though we have not addressed re-execution in detail this paper, the CRP model is easily compatible with our re-execution concept; we have worked on the default assumption that services are retriable, while pivot and compensatable services would be represented as additional constraints on the set of possible behaviors.

7 RELATED WORK

Current approaches for dealing with runtime faults in composite service execution either assume the existence of an independent formal specification in the Event Calculus (e.g., [5]) or an explicit definition of detailed fault handling logic, relying on predefinition of detailed fault models and explicit specification of exception handling strategies to be followed in a particular situation [8].

Dependency tracking techniques are well-known techniques for model-based diagnosis of programs [14] and Web Services [1].

Expressive constraint models have been developed to increase the accuracy of model-based debugging of imperative programs [9, 11]. While our processes are much simpler "programs", we cannot rely on the precise behavioral specification of the programs required by the earlier approaches. Instead, we exploit specific behavior instances observed at run time and embrace a symbolic representation to address the problem of incomplete information.

The repair planning problem in this paper, as it has been examined, e.g,. in [4, 12]. The former assumes the existence of a diagnosis method to initiate the planning process, but does not specify it; our work therefore complements [4] and provides a basis for application of its methods. The latter requires explicit definition of planning operators representing application semantics to enable use of AI planners.

8 CONCLUSION

In many practical diagnosis/repair scenarios where service process executions have to be repaired, only partial knowledge about the behavior of activities is available. In order to recover from failures, repair-enabled process execution engines apply sequences of executions and re-executions, possibly resulting in exceptions signaling failures. For the construction of correct repair plans without extensive separate specification requirements, the appropriate diagnosis methods are a necessary precondition.

In this paper we pointed out the limitations of classic dependency tracing methods for process diagnosis and motivated the necessity to reason with multiple possible activity behaviors including the propagation of symbolic values. We have proposed a diagnosis approach which (1) can deal with partial knowledge about activity behaviors and (2) does not assume that activities are executed in an order as defined in the process.

Both properties are necessary in diagnosis/repair scenarios where only limited behavior knowledge is available.

We presented a diagnosis method for process executions that relies only on observations gathered from concrete executions to infer possible faults in the execution of individual process steps. Our method can deal with partial information and non-deterministic activity behavior but requires only a structural model of the process and tolerates partially known behavior descriptions. We empirically confirmed the increased precision of our method and its feasibility for practical applications using a library of service processes. Thus our work lays the foundation for the overall goal of constructing complete and correct repair methods for processes where only partial behavior knowledge is available. We also address aspects of the repair process by seamlessly incorporating the repeated execution of activities.

This paper is an important first step towards a comprehensive diagnosis and repair framework for Web services. We intend to further explore the properties of our framework with respect to classical consistency-based and abductive diagnosis frameworks, and to further integrate the diagnosis method with recent work addressing the repair aspect of the problem [4].

Acknowledgements The work is partially funded by the Australian Research Council (ARC) under Discovery Project DP0881854 and the Austrian National Science Fund (FWF) Project 813806 - C2DSAS.

References

1. L. Ardissono et al. Enhancing web services with diagnostic capabilities. In *European Conference on Web Services*, 2005.
2. Fabio Casati, Stefano Ceri, Stefano Paraboschi, and Giuseppe Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Trans. Database Syst.*, 24(3):405–451, 1999.
3. Ron Cytron et al. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, 1991.
4. G. Friedrich, M. Fugini, E. Mussi, B. Pernici, and G. Tagni. Exception handling for repair in service-based processes. *IEEE TSE*, 2010.
5. Walid Gaaloul, Sami Bhiri, and Mohsen Rouached. Event-based design and runtime verification of composite service transactional behavior. *IEEE TSC*, 3(1):32–45, 2010.
6. James C. King. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.
7. D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *VMCAI*, volume 2575 of *LNCS*, pages 298–309, 2003.
8. An Liu, Qing Li, Liusheng Huang, and Mingjun Xiao. Facts: A framework for fault-tolerant composition of transactional web services. *IEEE TSC*, 3(1):46–59, 2010.
9. Wolfgang Mayer and Markus Stumptner. Evaluating models for model-based debugging. In *Proc. ASE*, pages 128–137. IEEE, 2008.
10. Sharad Mehrotra, Rajeev Rastogi, Henry F. Korth, and Abraham Silberschatz. A transaction model for multidatabase systems. In *ICDCS*, pages 56–63, 1992.
11. M. Nica, J. Weber, and F. Wotawa. How to debug sequential code by means of constraint representation. In *Proc. DX Workshop*, 2008.
12. D. Rao, Z. Jiang, and Y. Jiang. Fault tolerant web services composition as planning. In *Proc. Int'l Conf.Intelligent Systems and Knoweldge Eng. (ISKE'07)*, 2007.
13. R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 23(1):57–95, 1987.
14. F. Wotawa. On the relationship between model-based debugging and program slicing. *Artif. Intell.*, 135(1-2):125–143, 2002.