Diagnosing Process Trajectories Under Partially Known Behavior

Gerhard Friedrich¹, Wolfgang Mayer², Markus Stumptner²

¹ Alpen-Adria Universität, Klagenfurt, Austria
gerhard.friedrich@uni-klu.ac.at
² University of South Australia, Adelaide, SA, 5095, Australia
mayer@cs.unisa.edu.au
mst@cs.unisa.edu.au

ABSTRACT

Diagnosis of process executions is an important task in many application domains, especially in the area of workflow management systems and orchestrated Web Services. If executions fail because activities of the process do not behave as intended, recovery procedures re-execute some activities to recover from the failure. We present a diagnosis method for identifying incorrect activities in 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 logic programming techniques. Our evaluation demonstrates that the framework yields superior accuracy to dependency-based methods on realistically-sized examples.

1 INTRODUCTION

The spread of orchestrated Web Services increased the importance of diagnosing errors in process executions. Orchestrated Web Services define a process where individual activities are implemented by Web Services. If individual activities fail during execution, repair must be carried out (Friedrich et al., 2010). 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 in terms of activity executions from observed execution traces.

While powerful techniques based on discrete event systems have been developed for process diagnosis (Pencolé and Cordier, 2005), virtually all correct and complete diagnosis methods introduced so far assume that a detailed model of the activities’ behavior is available while some of the processes’ transitions and variables are unobservable. Unfortunately, this assumption does not hold in the Web Service process and workflow context. In this domain, the correct control flow is 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.

To solve the problem of partial knowledge, earlier work has used dependency tracing (Wotawa, 2002; Yan et al., 2009; Ardissono et al., 2005). 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 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. Rather, partial models of activities are gathered from observed execution traces.

Following consistency-based diagnosis (Reiter, 1987), 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.

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 OR-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 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.

Assume that the process was executed as shown in Fig. 1. Time points mark the end of an executed activity. The completion of execution activities 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 SAMPLEt1 and SEC2t4 are the only ones that could have failed. ⟨SAMPLEt1, SEC2t4⟩ is the only minimal conflict so far; a correctness assumption of SEC1 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 SAMPLEt1 and SEC2t4 are assumed to be correct. Diagnosis methods based on tracing dependencies (Wotawa, 2002) would not exonerate SEC1t2 since the computation of E depends on the output of SEC1t2.

Let us assume that a failure of SAMPLEt1 is unlikely, so [SEC2t4] is the only leading diagnosis. In this case, it follows, that SEC2 must output to R2 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 (Friedrich et al., 2010) decides to execute REM and GUARD after the execution of GUARD at t8, but the execution of GUARDt10 generates another exception. It follows that ⟨SAMPLEt1, REMt9⟩ is a further minimal conflict. Whatever branch is taken in the process, assuming executions SAMPLEt1, REMt9 as correct leads to an exception raised by GUARD. Consequently, ⟨SAMPLEt1⟩ 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 [SAMPLEt1] has to be extended to [SAMPLEt1, SAMPLEt11].

Consequently, there are two minimal diagnoses [SAMPLEt1, SAMPLEt11] and [SEC2t4, REMt9]. 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 REMt9. Next GUARD is executed which returns t (true). At this point we can conclude that the value provided by REMt12 corresponds to a value where the process is executed for the input provided by STARTt10 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.

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 (Nica et al., 2008) 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 (Reiter, 1987), 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 $P = (A, V, I, O)$ 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 $V$. $I \subseteq V$ and $O \subseteq V$ represent the input and output variables of $P$, respectively.

Activities $A_i$ may occur several times in the process exploiting different process variables. Occurrence $j$ of activity $A_i$ is denoted by $A_i^j$. Each occurrence $j$ of activity $A_i$ in $P$ receives input 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 $\tilde{I}_i$ ($\tilde{O}_i$). 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 (Cytron and others, 1991), where each variable is defined by exactly one activity. This is accomplished by creating new indexed “versions” of variables and by introducing so called $\phi$-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

\[ P(\mathcal{V}) = \bigwedge A_{i_j}(\mathcal{I}_{i_j}, \mathcal{O}_{i_j}), \]

\[ A_{i_j} \in A; \mathcal{I}_{i_j}, \mathcal{O}_{i_j} \subseteq \mathcal{V}; i \in [1, n] \]

that defines the control and data flow of the process. We use upper case letters to denote variables in first-order logical sentences. We write \( P(\mathcal{I}, \mathcal{X}, \mathcal{O}) \) to denote the conjunction \( P(\mathcal{V}) \) where the process receives input values assigned to \( \mathcal{I} \), assigns output values to \( \mathcal{O} \), and assigns values to process-internal variables in \( \mathcal{X} \). Symbol \( A_i \) denotes a relation that governs the allowed value combinations admitted by the correct behavior of all occurrences of activity \( A_i \). This is called the behavior relation and will be defined below. Hence, value assignments to all process variables \( \mathcal{I}, \mathcal{X}, \mathcal{O} \) which satisfy the relations of the activities \( A_i \) in the conjunction \( P(\mathcal{I}, \mathcal{X}, \mathcal{O}) \) correspond to the allowed execution(s) \( P \) where \( P \) receives input values \( \mathcal{I} \) and produces output values \( \mathcal{O} \). A value assignment that satisfies all relations \( A_i \) in \( P(\mathcal{I}, \mathcal{X}, \mathcal{O}) \) 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 form of the example process is represented as the conjunction

\[ P(\text{SPEC}, R_1, R_2, S_1, \ldots, S_4, C_1, C_2, \ldots, E) = \]

\[ \text{start}(C_1, \text{SPEC}) \land \text{sample}(C_1, \text{SPEC}, C_2, S_1) \land \]

\[ \text{sec1}(C_2, S_1, C_3, R_1) \land x1(C_3, R_1, C_4, C_5) \land \]

\[ \text{sec2}(C_4, S_1, C_6, R_3) \land x1(C_6, R_2, C_7, C_8) \land \]

\[ \text{rem}(C_7, S_1, C_9, S_2) \land \phi(C_8, C_9, C_{10}) \land \phi(S_1, S_2, S_3) \land \]

\[ \phi(C_5, C_{10}, C_{11}) \land \phi(S_1, S_3, S_4) \land \text{guard}(C_{11}, S_4, E) \land \text{end}(E, S_4) \]

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 \( \phi \)-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 \( \gamma \) may be bound to different process variables \( \gamma' \) 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 \( U_1, \ldots, U_n \) where the input variables are \( T = \langle U_1, \ldots, U_n \rangle \) and the output variables are \( O = \langle U_{n+1}, \ldots, U_s \rangle \), and let \( D_{U_j} \) denote the value domain of variable \( U_j \). The allowed behavior of activity \( A \) is given as a relation over the allowed input and output values: \( A(T, O) \subseteq D_{U_1} \times \cdots \times D_{U_s} \times D_{U_{n+1}} \times \cdots \times D_{U_s} \).

We require that \( A \) is total, that is, \( A(\mathbf{v}, O) \) includes at least one tuple for each \( \mathbf{v} \in D_{U_1} \times \cdots \times D_{U_s} \). 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 \( \langle A_1(T_1, O_1), \ldots, A_n(T_n, O_n) \rangle \). \( T_i, O_i \) denote vectors of activity variables.

To accurately model the flow of control in a process execution, we assume that each domain \( D_{U_j} \) contains a distinguished symbol \( \text{nil} \) 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 OR-join are defined as usual where control input and output variables have the binary domain \( \{ t, \text{nil} \} \). For processing activities (those which process inputs and pass the control flow), guards, and XOR-splits, we amend the relation \( A(T, O) \) to include all tuples \( \langle \mathbf{v}, \text{nil}, \ldots, \text{nil} \rangle \) where an input value in \( \mathbf{v} \) is \( \text{nil} \) and all other input variables are bound to values of their domain. For \( \phi \)-activities the output is \( \text{nil} \) iff both inputs are \( \text{nil} \). We refer to these sets of tuples as the \( \text{nil}\)-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 \( \text{nil} \) 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.
An element \( B_P \) of \( B_P \) is a possible process behavior.

Assume an execution of \( P \) results in the following observed behavior of activities \( \text{Obs} \):

\[
\begin{align*}
\{ & \text{start}(t, \text{spec}1), \text{sample}(t, \text{spec}1, t, s1), \text{sec}1(t, s1, t, r11), \\
& \text{x1}(t, r11, t, nil), \text{sec}2(t, s1, t, r2), \text{x1}(t, r2, nil, t), \ldots, \\
& \text{guard}(t, s1, nil), \text{rem}(s1, t, s2), \text{guard}(t, s2, nil), \\
& \text{sample}(t, \text{spec}1, t, s1), \text{rem}(s1, t, s3), \text{guard}(t, s3, t) \}.
\end{align*}
\]

The same I/Os are observed for the executions of \( \text{SAMPLE} \), while \( \text{REM} \) produces different outputs for the same input.

In absence of further information, the observed execution behaviors in \( \text{Obs} \) together with the \( \text{nil} \)-description comprise the behavior relations. Behavior relations of \( \phi \)-activities and END are also included.

Assume that \( \text{REM} \) may behave non-deterministically for some inputs, and that for the input value \( r2 \), the behavior of the XOR is unknown; that is, no behavior matching \( x1(t, r2, \ldots) \) has been observed. Then there are two possible process behaviors \( B^U_1 \) and \( B^U_2 \) for \( P \):

- For the second occurrence of \( x1 \) in \( P \) activates the upper branch on input \( r2 \), while in \( B^U_2 \) the lower branch is taken.

A given process behavior \( B_P \in B_P \) 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 \( P = (A, V, I, O) \). An assignment of value \( w \) to output variable \( Q \in O \) is reachable under \( B_P \) iff some execution admitted by \( P(\overline{I}, \overline{X}, \overline{O}) \) satisfies \( Q = w \). We write

\[
B_P \models \exists I X \overline{O} : P(\overline{I}, \overline{X}, \overline{O}) \wedge Q = w.
\]

For the scenario described above it holds that in both possible process behaviors \( B^U_1 \) and \( B^U_2 \)

\[
E = \text{nil} \quad \text{is a reachable assignment:} \quad B^U_1 \models P(\text{SPEC}, R_1, R_2, S_1, \ldots, S_4, C_1, C_2, \ldots, E) \wedge E = \text{nil} \quad (\text{the variables of } P \text{ are existentially quantified}).
\]

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 \( B^U_1 \).

If \( B^U_2 \) determines the execution, \( E = \text{nil} \), because the guard signals an exception if \( s_4 = s_2 \) is reached. If \( B^U_2 \) is changed to \( B^U_2 ' \) by removing \( \text{rem}(t, s_1, t, s_2) \) from the behavior relation of \( \text{REM} \), \( E = \text{nil} \) is no longer reachable in \( B^U_2 ' \), but is still reachable in \( B^U_2 \).

The process behavior \( B^U_2 \) specifies a process where regardless of the concrete execution—no exception will be raised, whereas \( B^U_2 ' \) admits an execution that fails. Consequently, if we assume that \( \text{SEC2}_{s4} \) produces a different value than the observed value \( r2 \) and on this value the upper path of the second occurrence of \( x1 \) is taken, and \( \text{REM} \) 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 \( A(\overline{I}, \overline{O}) \) 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 \( A_i(\overline{I}, \overline{O}) \) is defined to include the \( \text{nil} \)-description, all observations gathered from executions of activities, and possibly other known concrete I/O behaviors. This set of predefined behaviors is denoted by \( \text{Pre} \). 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 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 $A_i(T_i, O_i)$ specifying which value combinations for activity variables $T_i, O_i$ are allowed in the behavior relation. For our purposes it is sufficient if $Re_i$ refers to known domain values of $T_i, O_i$. 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 $DP = (P, Obs, Pre, Re)$ 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 $Obs = \{ob_1(w_1, w_2), \ldots, ob_n(w_1, w_2)\}$ be the set of observed I/O behaviors of activity executions, where $ob_i(w_1, w_2) \in Obs$ is the observed execution of an activity occurrence $A_i$. The set of all observed execution behaviors of an activity $A_i$ is denoted by $ob_i \subseteq A_i(T_i, O_i), A_i(T_i, O_i) \in Pre$ 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 $\{x1(t, nil, nil, nil), x1(nil, X, nil, nil) \mid X \in DP\} \cup Obs$ where $Obs$ contains the observations $\{x1(t, r_1, t, nil), x1(t, r_2, t, nil, t)\}$. The requirements $Re_{X1}$ for the XOR behavior are given by the following sentence:

\[
x1(C, W, Y, N) \Rightarrow \begin{cases} \text{nil} & \text{if } E = \text{nil} \text{ or } W = \text{nil} \\ \text{nil} \wedge N = \text{nil} \lor \text{Y = nil} & \text{if } C = \text{nil} \wedge W \neq \text{nil} \\ \text{Y = nil} \wedge N = \text{nil} & \text{if } C \neq \text{nil} \wedge W \neq \text{nil} \end{cases}
\]

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 $\Delta$, $B_P \setminus \Delta$ is the process behavior where from each behavior relation in $B_P$ the tuples of $\Delta$ are removed.

**Definition 7 (Diagnosis)** Let $DP = (P, Obs, Pre, Re)$ 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 $A_i(T_i, O_i) \in B_P$ is a superset of $A_i(T_i, O_i) \setminus \Delta$ for $A_i(T_i, O_i) \in Pre$
2. Each $A_i(T_i, O_i) \in B_P$ is consistent with $Re_i \in Re$
3. $B_P \not\exists \exists X \in P(\emptyset, X, E) \wedge E = \text{nil}$

$\Delta$ 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 last condition states that the diagnosed and repaired execution must be error free; that is, no execution must imply the failing of a guard.

Hence, a diagnosis $\Delta$ 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 $\Delta$ 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, spec1, t, s1)) 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 $E = \text{nil}$ is not a diagnosis. If $\Delta = \{\text{sample}(t, \text{spec1}, t, s_1)\}$ 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 $\Delta = \{\text{sec1}(t, s_1, t, r_1)\}$ 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, s1 will be assigned to S1 and therefore the guard will output nil (as in the
original execution). Thus, $\Delta = \{ \text{sec}1(t, s_1, t, r_{11}) \}$ 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). 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 (Kroening and Strichman, 2003).

5 SYMBOLIC REPRESENTATION
To verify if $\Delta$ is a diagnosis, behavior relations $A_i'(\overline{T}_i, \overline{O}_i)$ of activities $A_i \in A$ must be found that include the tuples of $A_i'(\overline{T}_i, \overline{O}_i) \setminus \Delta$, are consistent with $Re_i$, and no guard fails, i.e. $E = \text{nil}$ cannot be reached. If no such set of behavior relations exist then $\Delta$ 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 $A_i'(\overline{T}_i, \overline{O}_i) \setminus \Delta$ 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 (King, 1976) 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 $\{r_{11}, r_{21}\} \subseteq D$. Both SEC1 and SEC2 may produce symbolic values $y_1$ and $y_2$ 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 $\Delta \subseteq \text{Obs}$. Let $A$ be an activity with input variables $\mathcal{T} = \langle U_1, \ldots, U_k \rangle$ and denoted the domain of variable $U_i$. The set of all input vectors of an activity is $w_T = \{ w \in D_{U_1} \times \cdots \times D_{U_k} \}$. Likewise the set of all output vectors of an activity is $w_O = \{ w \in D_{O_1} \times \cdots \times D_{O_k} \}$. 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.

$$A_i'(\overline{T}_i, \overline{O}_i) = \{ \bigcup_{w \in w_T} \{ (w_1, w_1), \ldots, (w_1, w_p) \} \}$$
$$w_1 \in w_{\overline{T}_1}, \ldots, w_p \in w_{\overline{T}_p}$$

All possible behavior descriptions in $A_i'$ are extended by the set of tuples considered to be correct, i.e. $A_i'(\overline{T}_i, \overline{O}_i) \setminus \Delta$ for $A_i'(\overline{T}_i, \overline{O}_i) \in \text{Pre}_i$ and by the nil-description. In addition, we eliminate all behavior descriptions in $A_i'$ 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 $\phi$-activities. The result is a set of possible process behaviors $B_P$ for a diagnosis problem and a $\Delta \subseteq \text{Obs}$.

For example, the domain $D_X$ of $x(C, W, Y, N)$ is extended to contain all of $\{r_{11}, r_{21}, y_1, y_2\}$ (and nil as the only other value). Every possible behavior of X1 includes the tuples in $\text{Pre}_X1$ (shown earlier) and must be consistent with $Re_X1$. That is, the behavior on inputs $\{r_{11}, r_{21}, \text{nil}\}$ is fixed, but there are four different behaviors which differ just on the outputs provided for inputs $\{y_1, y_2\}$:

$$\{x1(t, y_1, t, \text{nil}), x1(t, y_2, t, \text{nil})\},$$
$$\{x1(t, y_1, \text{nil}, t), x1(t, y_2, \text{nil}, t)\},$$
$$\{x1(t, y_1, \text{nil}, t), x1(t, y_2, \text{nil}, t)\},$$
$$\{x1(t, \text{nil}, t), x1(t, \text{nil}, t)\}.$$
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 $B_P$. It follows that if there is a process behavior $B_P$ for which $\text{Pre} \setminus \Delta$ can be correctly extended, then there exists a process behavior $B_P'$ in the set of generated possible process behaviors $B_P$ which is also a correct extension of $\text{Pre} \setminus \Delta$. We have constructed a decision method which correctly and completely determines if a set $\Delta \subseteq \text{Obs}$ is a diagnosis. □

Given our example process, observations and the diagnosis candidate $\Delta = \emptyset$, all $A_i(T_i, \overline{O}_i)$ in the process behaviors of the generated set $B_P$ contain their observed execution behavior. Therefore, in each process behavior of $B_P$, $E = \text{nil}$ is reachable. Consequently, there is no $B_P'$ which is a correct extension of $\text{Pre} \setminus \Delta$. Hence, $\Delta = \emptyset$ is not a diagnosis. However, if the observed execution behavior $\text{sample}(t, \text{spec}_t, t, s_1)$ is removed, then only symbolic (unseen) output values remain to be assigned to $S_1$. Therefore, we can construct behaviors for $\text{REM}$, $\text{GUARD}$, and all other activities such that the guard is not triggered for any execution. Therefore, $\{\text{sample}(t, \text{spec}_t, t, s_1)\}$ 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 $E = \text{nil}$ 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 (Friedrich et al., 2010), and generated additional (artificial) processes to obtain a comprehensive 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.

Our results are summarized in Table 1, aggregated by process size. The columns show the average number of activities in a process ($\text{Size}$), the number of process executions considered in our study ($\text{N}$), the average number of decision nodes in a process ($\text{Xor}$), the average number of activities in each execution trace ($\text{Trace}$), the number of minimal diagnoses obtained from dependency-based models ($\text{Dep}$) and from our model ($\text{Symb}$), the mean relative improvement ($\text{Imp} = 1 - \text{Symb/Dep}$), and the average diagnosis time in seconds ($\text{Time}$) 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 $\text{Imp}$ may be larger than $1 - \text{Symb/Dep}$ 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, $\text{Imp}$ 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

Table 1: Comparison of dependency- and symbolic diagnosis model

<table>
<thead>
<tr>
<th>Size</th>
<th>N</th>
<th>Xor</th>
<th>Trace</th>
<th>Dep</th>
<th>Symb</th>
<th>Imp</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–9</td>
<td>48</td>
<td>0.67</td>
<td>8.04</td>
<td>3.98</td>
<td>5.00</td>
<td>0.10</td>
<td>0.12</td>
</tr>
<tr>
<td>10–19</td>
<td>100</td>
<td>3.10</td>
<td>16.84</td>
<td>6.40</td>
<td>4.63</td>
<td>0.16</td>
<td>1.07</td>
</tr>
<tr>
<td>20–29</td>
<td>124</td>
<td>5.21</td>
<td>27.16</td>
<td>8.97</td>
<td>4.89</td>
<td>0.45</td>
<td>5.43</td>
</tr>
<tr>
<td>30–39</td>
<td>60</td>
<td>7.67</td>
<td>33.43</td>
<td>9.67</td>
<td>5.37</td>
<td>0.59</td>
<td>16.85</td>
</tr>
<tr>
<td>40–49</td>
<td>28</td>
<td>10.21</td>
<td>41.64</td>
<td>12.36</td>
<td>6.31</td>
<td>0.50</td>
<td>74.52</td>
</tr>
<tr>
<td>50–59</td>
<td>12</td>
<td>12.17</td>
<td>35.17</td>
<td>10.42</td>
<td>6.29</td>
<td>0.32</td>
<td>124.22</td>
</tr>
<tr>
<td>60–69</td>
<td>16</td>
<td>14.62</td>
<td>30.75</td>
<td>9.12</td>
<td>4.50</td>
<td>0.40</td>
<td>130.63</td>
</tr>
<tr>
<td>70–79</td>
<td>6</td>
<td>17.00</td>
<td>35.67</td>
<td>9.33</td>
<td>2.00</td>
<td>255.80</td>
<td></td>
</tr>
<tr>
<td>0–79</td>
<td>400</td>
<td>5.85</td>
<td>24.95</td>
<td>8.19</td>
<td>4.82</td>
<td>0.29</td>
<td>14.56</td>
</tr>
</tbody>
</table>

1The data were obtained from Eclipse 6.1 on Intel P4@1.86GHz with 6Gb RAM running Linux 2.6.
of activities is virtually always 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.

7 RELATED WORK
Dependency tracking techniques are well-known techniques for model-based diagnosis of programs (Wotawa, 2002) and Web Services (Ardissono et al., 2005). The approach has been employed to diagnose Web service workflows built from BPEL models (Yan et al., 2009). The semantics of the service are expressed as an automaton and diagnosis is conducted by tracing backwards along the dependencies when a fault occurs. These mechanisms correspond to the dependency-based approach in our evaluation.

Expressive constraint models have been developed to increase the accuracy of model-based debugging of imperative programs (Mayer and Stumptner, 2008; Nica et al., 2008). 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.

Related work on planning for diagnosis (McIlraith, 1998) has been concerned with fault explanation by constructing sequences of events that lead to a particular set of observations given a set of operations. In contrast, we assume that the process structure is given and fixed but the effects of each activity may not be entirely known.

Logging mechanisms facilitating compensation of failing transactions have also been analyzed (Biswas, 2004). However, the paper focuses on sequential compensation and does not identify diagnoses.

8 CONCLUSION
In this paper we pointed out the limitation of dependency tracing methods and motivated the necessity to reason with multiple possible activity behaviors including the propagation of symbolic values. We 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 introduced symbolic values to deal with unknown behavior and obtained a complete and correct diagnosis method, provided a correct and complete model of the known activity behavior and the constraints on the possible behaviors are given. In our evaluation we empirically confirmed the increased precision of our method and its feasibility for practical applications.

Acknowledgements The research project is partially funded by the Austrian Research Promotion Agency (Project 813806 - C2DSAS) and by the Australian Research Council (Grant DP0881854).

REFERENCES