Functional Diagnosis of a SOA’s BPEL Processes

Birgit Hofer and Seema Jehan and Ingo Pill and Franz Wotawa
Institute for Software Technology, Graz University of Technology, Austria

Abstract

The complexity of software products driving today’s society, e.g., in the context of clouds, certainly asks for diagnostic reasoning that helps us in their design and maintenance. We aim to contribute to answering this need by proposing a model-based approach at functional diagnosis of a SOA’s BPEL processes. These processes implement desired goals by orchestrating individual web services through a specified business logic. Partial knowledge about the invoked services’ details is just one of the challenges that we have to face in a diagnostic process. For our consistency-oriented reasoning, we extract a BPEL flow graph, and annotate it with pre- and postconditions on individual system parts. The actual diagnoses are derived from a corresponding constraint representation. First experiments showed promising results regarding our approach’s viability.

1 Introduction

The principle idea of model-based diagnosis is to consider a system’s model for detecting deviations between a system’s observed and expected behaviors, and in turn isolate possible root causes for detected issues. A software product’s inherent complexity makes this a cumbersome task for corresponding systems. Moreover, as discussed by Friedrich et al. in [1], this becomes an even more demanding challenge, if we have to consider service oriented architectures (SOAs). That is, complementing controllability and observability issues, we have to deal with the fact that usually we have only partial knowledge about the invoked web services’ functionality. SOAs, however, have crystallized as a promising concept for future computing [2]. The underlying concept of encapsulating certain functionality and products (IP cores) via web services for flexible and lightweight network access, provides a unique and flexible environment. At the cost of requiring permanent network access, it allows us to implement complex software by dynamically orchestrating web services we consider to fit our purposes best.

In [3], we envisioned a corresponding, combined diagnosis and testing workflow for SOAs that targets its BPEL processes. BPEL emerged almost a decade ago as an OASIS¹ standard for modeling (and executing) the business logic of processes in order to orchestrate web services for achieving a desired goal. Our particular focus is on synchronous BPEL process variants. As reasoning model, we extract from such a process a BPEL flow graph (see Section 2) that contains all its internal data, and which gets annotated with all the information (which is most likely still partial) about the invoked web services. That is, we add corresponding pre- and postconditions, and support also the annotation with further side information available.

Regarding a classification, one could consider our reasoning to be a white-box approach, as we derive our reasoning model from the BPEL process directly. As quite a portion of such a system’s functionality lies in the invoked web services (for which only partial knowledge is available), we consider the term grey-box to be more appropriate. That is, we focus on a BPEL process’ behavior when “integrating” all accessed parts, and not on the isolated business logic.

While we have focused on the testing issue so far [3; 4; 5; 6], diagnostic reasoning support for unexpected results (like failing test cases) would be a highly welcomed asset for such complex systems. In this paper, we report on our first steps in this direction and show in Section 3 how to isolate possible faults in the BPEL process for failing test cases, i.e., whenever we encounter unexpected behavior. Similar to the approach Wotawa et al. proposed for debugging Java programs [7], we rely on a constraint representation and implement a consistency-oriented model-based diagnosis approach. In particular, we derive a constraint representation for a BPEL flow graph, and then compute diagnoses using a constraint solver. For the constraint representation, we do not consider the entire BPEL flow graph. Instead, we propose a more focused approach which translates only one concrete path into constraints. This allows for providing attractive run-times in practice.

There has been work focusing on aspects of self-healing web services, the WS-Diamond project². Ardagna et al. [8] described a framework for adaptive web-service processes. Yan and colleagues [9] introduced a model-based diagnosis approach for web-services. Trave-Massuyes et al. [10] considered the question of how to define diagnosability of systems and its use for diagnosing web-services. Ardissono et al. [11] discuss the question of enhancing web-service compositions using diagnosis. In contrast to these papers, we describe a solution that combines trace analysis with constraint solving for fault localization with the goal of improving the overall diagnosis performance.

¹Organization for the Advancement of Structured Information Standards, see https://www.oasis-open.org

²http://wsdiamond.di.unito.it/
For evaluation purposes (see Section 4), we specifically considered the concept of program mutation. Like we did for our test suite performance evaluation in [6], we used a mutation tool [12] to derive faulty “program” versions. For these mutants we derived test cases that would trigger different observable behavior (in form of the output values) for one and the same input values for the original and mutated programs. Then we verified whether our diagnostic reasoning could identify the mutation, i.e., provide a corresponding diagnosis, given the incorrect program, as well as the input and the expected output values. For the experiments reported in Section 4, we considered examples with injected single faults only.

We illustrate our approach by means of the Triangle example shown in Figure 1. The corresponding program takes the triangle’s sides’ lengths as input, and classifies the triangle to be equilateral, isosceles, or scalene. The example is well-known in the software engineering domain, and can be easily modeled in BPEL so that it perfectly suits the requirements of a running example.

![BPEL Flow Graph](image)

**Figure 1:** BPEL Flow Graph of the Triangle Example Process

The corresponding BPEL flow graph is illustrated in Figure 1. First, the input values are assigned, i.e., the lengths of the triangle’s sides a, b, and c. Afterwards, with \( \text{CondA} = (a > 0) \land (b > 0) \land (c > 0) \) we check whether those values are indeed positive numbers. If not, a corresponding error message is returned (“\( \text{NEG} \)”, where the program’s actual output is “No triangle”). Otherwise, we invoke the web service \( \text{InvokeValid} \), in order to check the triangle for inequality. That is, this web-service determines whether the given lengths indeed allow for a valid triangle such that the constraints \( (a + b) > c \), \( (a + c) > b \), and \( (b + c) > a \) are satisfied. The pre-condition for this service we could derive from \( \text{CondA} \), i.e., assume that we have \((a > 0) \land (b > 0) \land (c > 0)\). As post-condition we could use \( \text{ret} = ((a + b) > c) \land ((a + c) > b) \land ((b + c) > a) \). Assigning the returned value to \( \text{CondB} \), we evaluate the data and return a corresponding error message if \( \text{CondB} \) is false (“\( \text{NO} \)”, where the program’s output is “Not a valid triangle”). Otherwise, we check if \( \text{CondC} = (a == b == c) \) whether a, b, and c form an equilateral triangle, s.t. \( a == b == c \). If not, \( \text{CondD} = ((a == b) \lor (a == c) \lor (b == c)) \) checks if the triangle is isosceles, that is, if we have either \( a == b \), \( a == c \), or \( b == c \). If contradicted, the triangle is classified as scalene.

For our running example, let us assume that we want to classify the isosceles triangle such that \( a=2 \), \( b=2 \), and \( c=1 \). Unfortunately and unexpectedly, however, our “faulty” program returns an error message instead (“\( \text{NEG} \)”, respectively “No triangle”), due to a programming error. That is, for \( \text{condA} \) we wrote \((a_0 < 0) \lor (b_0 > 0) \lor (c_0 > 0) \) instead of \((a_0 > 0) \lor (b_0 > 0) \lor (c_0 > 0) \), such that the program takes the corresponding else branch for \( \text{condA} \) due to the condition evaluating to false instead of true for our example. In order to isolate possible root causes for this issue, our approach extracts the depicted BPEL flow graph, in order to then identify those constraints encountered along the path that can explain the encountered inconsistency (why the test failed).

## 2 Definitions

Our central reasoning concept is based on a BPEL flow graph [4] that we extract from a BPEL process. We use the labels \( \gamma_C(v) \) and \( \gamma_A(v) \) of a node \( v \) for adding our partial knowledge about web services and other side information.

**Definition 1.** A BPEL Flow Graph \( G \) is defined by a tuple \( G = (V, E, v_0, F, \gamma_C(v \in V), \gamma_A(v \in V)) \), where \( V \) is a finite set of vertices representing BPEL process activities, \( E \subseteq V \times V \) is a finite set of directed edges representing the connections between BPEL activities \( (v_1 \rightarrow v_2) \in E \) connects \( v_1 \) to \( v_2 \), \( v_0 \in V \) is the start vertex, \( F \subseteq V \) is the set of leaf vertices (with no outgoing edges), and the functions \( \gamma_C(v) \) and \( \gamma_A(v) \) map vertices \( v \in V \) to activity conditions and assignments respectively.

With \( G \) capturing the control flow structure of the implemented business logic, a path in \( G \) defines a valid scenario.

**Definition 2.** A finite Path \( \pi \) of length \( n \) in a BPEL Flow Graph \( G = (V, E, v_0, F, \gamma_C(v \in V), \gamma_A(v \in V)) \) is a finite sequence \( \pi = \pi_1 \pi_2 \ldots \pi_n \) such that we have that (1) for any \( 0 < i \leq n \), \( \pi_i \) is in \( V \), (2) \( \pi_1 = v_0 \), (3) for any \( 0 < i < n \), the edge \( e = (\pi_i, \pi_{i+1}) \) is in \( E \), and (4) \( \pi_n \in F \). The length of some path \( \pi \) is denoted by \( |\pi| \), where we use \( f(\pi) \) to refer to the last vertex in sequence \( \pi \). As we reason about finite computations only, per definition a path is finite.

For reasoning about a path \( \pi \), like determining its feasibility in general, we collect all the constraints along \( \pi \).

**Definition 3.** For a path \( \pi = \pi_1 \pi_2 \ldots \pi_n \) in some BPEL flow graph \( G = (V, E, v_0, F, \gamma_C(v \in V), \gamma_A(v \in V)) \), we define the set of Path Constraints \( C(\pi) = \bigcup_{0 \leq i < n} \gamma_A(\pi_i) \cup \gamma_C(\pi_i) \), where variables are replaced by indexed variables in order to implement a static single assignment form.

In detail, the static single assignment form (SSA) [13] means that we use indexed variables (i.e. “temporal” variable instances clocked by assignments), such that, whenever a variable is assigned a value, the index is incremented for further referrals along the path. This process ensures that every variable along a path is only defined once (but might be referenced many times).

**Example 1.** The path constraints for our running example are as follows:
Given the definitions of a path and its path constraints, we are able to specify feasible paths and traces.

**Definition 4.** A Feasible Path $\pi$ is a path as of Def. 2 such that its path constraints $\mathcal{C}(\pi)$ are satisfiable. A corresponding satisfying assignment defines a trace $\tau$ for path $\pi$. We denote the input part of $\tau$ as $I(\tau)$.

**Example 2.** Our running example’s input $I = \{a = 2, b = 2, c = 1\}$ results in the grey-shaded path as of Fig. 1. Accordingly the corresponding constraints are satisfiable and the path is feasible.

For assessing our approach’s viability, we consider its diagnostic performance for the scenario that the trace $\tau$ (the “observable” in- and output sequence) for some given (incorrect, i.e., in our case artificially mutated) program deviates from the expected behavior for input $I(\tau)$ (in our case given by the trace for the known unmutated program). Obviously this is a specific and artificial variant of the common scenario of a failing test case, and in turn of the abstract scenario of encountering some unexpected behavior.

**Definition 5.** A Mutant is an altered version $G'$ of an original program $G$. The mutant $G'$ is equivalent to $G$, if and only if their behavior does not differ s.t. the observable output for a given input is equal for $G$ and $G'$.

For deviating behavior, taking the unexpected trace and the corresponding path into account, the aim of our reasoning is to obtain a diagnosis (apparently its presence in a set of viable diagnoses) identifying the mutation.

**Example 3.** For our running example, the mutant $G'$ s.t. condition $\text{CondA} (a > 0) \land (b > 0) \land (c > 0)$ is replaced by $(a < 0) \land (b > 0) \land (c > 0)$ is not an equivalent mutant of $G$, as its input $I = \{a = 2, b = 2, c = 1\}$ triggers different traces for $G$ and $G'$.

For our diagnostic reasoning, we follow the concept of consistency-oriented model-based diagnosis as defined in [14; 15; 16]: Given a system’s set of components $\text{COMP}$, a system description $\text{SD}$ defining the correct behavior $\lnot \text{AB}(c_i) \Rightarrow \text{NominalBehavior}(c_i)$, and some actual observations $\text{OBS}$ about the system’s behavior, the system is considered to be at fault iff $\text{SD} \cup \text{OBS} \cup \{\lnot \text{AB}(c_i) | c_i \in \text{COMP}\}$ is inconsistent. The predicate $\text{AB}(c_i)$ represents the “health” state of component $c_i$. It is abnormal and we know nothing about its behavior, or it behaves as expected. While a minterm in the assumptions defines a specific state of the system, a diagnosis $\Delta$ is a subset-minimal set of faulty components that explains the inconsistency and is a subset of at least one “consistent” minterm. Our components will be the individual constraints.

**Definition 6.** A diagnosis for $(\text{SD, COMP, OBS})$ is a subset-minimal set $\Delta \subseteq \text{COMP}$ such that $\text{SD} \cup \text{OBS} \cup \{\lnot \text{AB}(c_i) | c_i \in \text{COMP} \setminus \Delta\}$ is consistent.

### 3 Behavioral Diagnosis of BPEL Processes

Based on our annotated BPEL flow graph, for an unexpected trace (e.g. a failed test case) the aim was to identify those program parts that could explain the deviation from the expected behavior. As software is certainly a complex domain, which is eliviated by our partial knowledge about the system’s parts (web services), we thus opted for an approach that allows us to identify solutions quite quickly due to a specific focus on certain system parts.

That is, in contrast to a full temporal system model like the one we used for our behavioral diagnosis of specifications in LTL [17], we focus on a specific path in the system’s control structure, similar to the idea in [18]. This way, the system description does not describe some sort of automation accommodating all the different branches in the system’s control structure, but a single sequence in SSA form. In particular, we use the path constraints $\mathcal{C}(\pi)$ for path $\tau$ in the incorrect program $G'$ as defined by $\tau$’s input values $I(\tau)$ as system description $\text{SD}$. Encapsulating $\mathcal{C}(\pi)$’s individual constraints $c$ with corresponding assumption predicates $\text{AB}(c)$, we aim at identifying those constraints $c$ that could be responsible for $\tau$’s deviation from expected behavior.

Faults in branching conditions (or related variables) might, however, lead to the fact that the path $\pi$ for $\tau$ in the incorrect program $G'$ deviates from the expected path in the repaired program $G$. With our focused system model, this would mean that the correct path $\pi_G$ could “fall out” of the system model at some branching point. Such a scenario requires us (a) to remove subsequent (in terms of temporal evolvement of the path) constraints in $\mathcal{C}(\pi)$ from the picture, and (b) to make the user aware of this. The latter is motivated by the fact that $\text{SD}$ does not provide information about subsequent behavior then, as everything aside path $\pi$ is considered as black box. To address these issues, we divide a trace $\tau$ into segments separated by the branching choices, and add variables $\text{intra}$, for all the segments. These variables are used to indicate whether a segment is part of the path described by a diagnosis or not. While for the first segment this would obviously not be necessary (it has to be in every scenario), a uniform treatment helps us in keeping the descriptions small.

**Definition 7.** A segmented path $(\pi, S)$ in $G$ is a tuple such that $\pi$ is a path, and $S$ is the set of branching points $s_i$ of $\pi$ in $G$ that divide the path into enumerated segments as follows. The branching points $s_i$ are numbered according to their distance from $s_0$, the enumeration starting with 1. The first segment, numbered 0 starts at $s_0$ and ends with $s_1$. Starting with segment 1, a segment $i$ starts right after $s_i$, and ends with either $s_{i+1}$ or the path’s end.

**Example 4.** The path constraints from Example 1 are divided into 2 segments. The first segments contains:

1. $a_0 = \text{in1}$
2. $b_0 = \text{in2}$
3. $c_0 = \text{in3}$
4. $(a_0 < 0) \land (b_0 > 0) \land (c_0 > 0)$ = false

The second segment contains:

5. $output_0$ = “No triangle”

Now let us define the specific diagnosis problem and our corresponding technical implementation.

**Definition 8.** A BPEL flow graph diagnosis problem is a tuple $(G', \tau)$, where $G'$ is a BPEL flow graph, and $\tau$ is an
observed trace that contradicts the system specification by showing unexpected behavior.

This problem describes the scenario where we have a program \( G' \) that we find to be incorrect by observing some unexpected behavior in the form of trace \( \tau \). For a consistency-oriented model-based diagnosis approach, we need to find a representation of the problem that we can use for an automated reasoning. To this end, we define the following encoding to be used with a constraint solver.

**Definition 9.** A constraint satisfaction encoding CSP\((G', \tau)\) for a BPEL flow graph diagnosis problem \((G', \tau)\) is defined as follows:

1. Let \( (\pi, S) \) be a segmented path in \( G' \) for \( I(\tau) \)
2. Let the set of variables \( V \) contain all the variables in \( C(\pi) \), as well as Boolean variables \( \text{intrac}_i \), for all of \( (\pi, S) \)'s segments as of Definition 7.
3. Let \( C'(\pi) \) be the path constraints \( C(\pi) \) altered s.t. for any individual constraint \( c \in C(\pi) \) of segment \( i \), we add a predicate \( \text{AB}_c \), and do as follows:
   - (a) if \( c \) is not a branching constraint from some \( s \in S \), then \( c \) gets replaced by \( \neg\text{intrac}_i \lor \text{AB}_c \lor c \).
   - (b) if \( c \) is the branching constraint from \( s_i \in S \), then \( c \) gets replaced by the following set of constraints:
     - \( \neg\text{intrac}_i \rightarrow \neg\text{intrac}_{i+1} \lor \text{AB}_c \lor \neg\text{intrac}_{i+1} \) and
     - \( \neg\text{intrac}_i \lor \text{AB}_c \lor (c \leftrightarrow \text{intrac}_{i+1}) \) - with \( \circ \) being the expected polarity of \( \text{intrac}_{i+1} \) when constraint \( c \) is satisfied.
4. Then let CSP\((G, \tau)\) be the combined constraints of \( C'(\pi) \) and \( \tau \) as well as the constraint \( \text{intrac}_0 \).

Now let us have a look at the corresponding CSP for our running example.

**Example 5.** The constraints as of Def. 9 for our running example are as follows:

- \( \neg\text{intrac}_0 \lor \text{AB}_1 \lor a_0 = \text{in1} \)
- \( \neg\text{intrac}_0 \lor \text{AB}_2 \lor b_0 = \text{in2} \)
- \( \neg\text{intrac}_0 \lor \text{AB}_3 \lor c_0 = \text{in3} \)
- \( \neg\text{intrac}_0 \rightarrow \neg\text{intrac}_1 \)
- \( \text{AB}_1 \lor \neg\text{intrac}_1 \)
- \( \neg\text{intrac}_0 \lor \neg\text{AB}_2 \lor (a_0 < 0) \land (b_0 > 0) \land (c_0 > 0) \leftrightarrow \neg\text{intrac}_1 \)
- \( \text{intrac}_0 \lor \text{AB}_3 \lor \text{output}_0 = "\text{No triangle}" \)
- \( \text{in1} = 2 \)
- \( \text{in2} = 2 \)
- \( \text{in3} = 3 \)
- \( \text{output}_0 = "\text{isosceles}" \)

In contrast to conflict-based computations like the one suggested by Reiter, we compute diagnoses directly from the model. To that purpose, and following the idea presented in [7] for Java programs, we formulate a corresponding constraint satisfaction problem adopting the CSP as of Definition 9 and derive diagnoses as satisfying assignments using a constraint solver such as MINION [19; 20]. Newer developments such as [21], and our experiments in [22] showed that such direct approaches might offer superior advantage. The MINION setup used for this paper was investigated as Direct-MS\(_{CS} \) in the context of our performance experiments with several setups as reported in [22].

Algorithm 1 illustrates the corresponding computation of diagnoses. The idea there is to take CSP\((G', \tau)\) and let the solver determine satisfying assignments that are limited in active abnormal predicates. Via adding the constraint in line 3 of the algorithm, we can effectively limit the sum of active abnormal predicates to the desired cardinality \( i \). A single query to the solver delivers all the solutions for a specified cardinality. Thus, starting with a cardinality of one, we increment the cardinality limit until we reach the given upper bound for the diagnosis cardinality. All solutions are stored (Line 5) and the constraints for the corresponding blocking clause (i.e. basically a logic or over the negated abnormal predicates in a diagnosis) are attached to the model, in order to ensure the diagnoses’ subset-minimality (Line 6).

**Algorithm 1 GetDiagnoses\((M, n)\)**

**Input:** A constraint system \( M \) and the upper bound of the diagnosis cardinality \( n \)

**Output:** All minimal diagnoses up to the predefined cardinality \( n \)

1. Let DS be \( \{\}\)
2. for \( i = 1 \) to \( n \) do
3. \( M' = M \cup \left\{ \left\lceil \sum_{j=1}^{n} \text{AB}_j \right\rceil == i \right\} \)
4. \( D = \text{Solve}(M') \)
5. \( DS = DS \cup D \)
6. \( M = M \cup \neg(\forall d \in D(d)) \)
7. end for
8. return DS

**Example 6.** Deploying algorithm 1 on the CSP as of Definition 9 for our running example results in the computation of the three single fault diagnoses \( \text{AB}_1, \text{AB}_3, \) and \( \text{AB}_5 \).

For a derived diagnosis, we propose that the user should be made aware of the different segments’ being part (or being excluded) of the scenario described by a diagnosis. For our running example, we show in Figure 2 the corresponding diagnoses where for those segments disabled by the corresponding intrac variables, for each constraint, the intrac variable is put in black and the remaining part of the constraint is shown in grey. As argued before and evident from the figure, this is vital data when considering the information of a diagnosis, so that we suggest to add to each diagnosis the corresponding minterm in the intrac variables, that is, the evaluation of all the corresponding intrac variables.

**Definition 10.** An extended diagnosis for a BPEL flow graph diagnosis problem \((G', \tau)\) as of Def. 8 is a tuple \((\Delta, \pi, \text{INTRACE})\), where \( \Delta \) is a diagnosis (see Def. 6) in the predicates \( \text{AB} \) for the CSP as of Def. 9 derived from the given BPEL flow graph diagnosis problem, \( \pi \) refers to the segmented path derived for CSP\((G', \tau)\), and \( \text{INTRACE} \) is the corresponding minterm in the intrac variables (cf. Def. 9) describing which segments are in the path depicted by \( \Delta \).

Please note that in our running example, we only had binary branching decisions. Therefore, we could identify the exact alternative branch in case of an abnormal branching constraint. For more complex constructs like a switch statement in C, obviously, the actual alternative branch cannot be automatically determined, so that the debugging process has to take this into account.
Figure 2: A graphical presentation of the running example’s single-fault diagnoses. Only the relevant constraints, and in turn their relevant parts, are highlighted, while all other parts are put in grey.

(a) Diagnosis \( AB_1 \): incorrect input value

\[
\text{RecInput} \rightarrow \text{AssignInput} \rightarrow \text{condA} \rightarrow \begin{cases} \text{then} & \text{in1} = 2 \\
\text{else} & \text{in2} = 2 \\
\end{cases} \\
\text{output}_2 = \text{“isosceles triangle”} \\
\text{intrac}_2
\]

(b) Diagnosis \( AB_4 \): incorrect branching constraint

\[
\text{RecInput} \rightarrow \text{AssignInput} \rightarrow \text{condA} \rightarrow \begin{cases} \text{then} & \text{in1} = 2 \\
\text{else} & \text{in2} = 2 \\
\end{cases} \\
\text{output}_2 = \text{“isosceles triangle”} \\
\text{intrac}_2
\]

(c) Diagnosis \( AB_5 \): incorrect output value

\[
\text{RecInput} \rightarrow \text{AssignInput} \rightarrow \text{condA} \rightarrow \begin{cases} \text{then} & \text{in1} = 2 \\
\text{else} & \text{in2} = 2 \\
\end{cases} \\
\text{output}_2 = \text{“isosceles triangle”} \\
\text{intrac}_2
\]

Figure 3: BPEL Flow Graph of the Bank Loan Business Process

Considering our encoding, it is apparent that we do not directly accommodate non-determinism in the program for scenarios where for one and the same input we could have different paths. However, for such scenarios one can apparently implement an exhaustive approach by considering all the options in a loop, and presenting the diagnoses grouped by the different choices.

4 First empirical experiments

For evaluating our approach, we built a corresponding prototype that extracts a BPEL flow graph from a corresponding process and translates it to MINION constraints. MINION (we used version 1.6.1) is an open source project, and supports arithmetic, logic, and relational operators over Boolean and Integer variables. All the experiments were carried out on a MacBook Pro (Late 2011) with a 2.4 GHz Intel Core i5, 4 GB 1333 MHz DDR3, running OS X 10.7.2.

In this paper, we report on our experiments with three programs. That is, complementing our running example (Triangle), we took also the SOA “programs” Bank Loan Process and BMI (as described in detail later on) into account. For creating faulty program versions, we used the Mubpel tool [12]. Then we searched for input values that would yield different traces for the mutant and the original program. While we considered single-fault injection only for the reported experiments, our approach is not limited to single faults in general and future experiments will cover scenarios with more than one mutated constraint.

In Figure 3 you can see the BPEL flow graph of the Bank Loan Process. Its main objective is to approve, delay, or reject loan requests based on the amount requested funds and the client’s history. Loan requests for less than 10,000 are approved immediately, if the client has a credible history (then there is a low risk involved in granting it). For all other requests, an external service (Assessor) is invoked.

In Figure 3 you can see the BPEL flow graph of the Bank Loan Process. Its main objective is to approve, delay, or reject loan requests based on the amount requested funds and the client’s history. Loan requests for less than 10,000 are approved immediately, if the client has a credible history (then there is a low risk involved in granting it). For all other requests, an external service (Assessor) is invoked.

The output of this external service would be one of “approved”, “pending”, or “reject”.

For scenario Loan-V1, we changed the IfLoan condition from \( \text{loan} \leq 10000 \) to \( \text{loan} \neq 10000 \). The test case \( t \) with \( I = \{ \text{loan} = 10000, \text{clientId} = 20 \} \) as input and the expected output \( E = \{ \text{reject} \} \) unveils this fault, as the mutant results in the output \( O = \{ \text{approved} \} \). That is, the mutated program takes a different branch and calls the “Risk” web service rather than the “Assessor” service, resulting in the actual output \( O = \{ \text{approved} \} \).
For Loan-V2, the $\text{IFLoan}$ condition was changed from $\text{loan} \leq 10\,000$ to $\text{loan} > 10\,000$. For input $I = \{\text{loan} = 1\,000\,000, \text{clientId} = 20\}$, the expected output was $E = \{\text{reject}\}$, but the mutated program returned $O = \{\text{approved}\}$. Again the different branch chosen by the program results in the actual output would be $O = \{\text{approved}\}$ derived by a different assessment variant. For scenario Loan-V4, we changed the $\text{IFLoan}$ condition from $\text{loan} \leq 10\,000$ to $\text{loan} \leq 100\,000$. The test case with input $I = \{\text{loan} = 1\,000\,000, \text{clientId} = 20\}$ was expected to take the “Else” branch and based on the loan amount give the output $E = \{\text{pending}\}$. But, since the $\text{IFLoan}$ condition was mutated, the “Then” branch was taken and the “Risk” web service derived the actual output $O = \{\text{approved}\}$. Mutating the $\text{LowRisk}$ condition from $\text{risk} = 0$ to $\text{risk} = 1$ yielded the mutated program for scenario Loan-V5. The test case $t$ with input $I = \{\text{loan} = 1\,000, \text{clientId} = 21\}$, was expected to yield output $E = \{\text{pending}\}$, but the mutated $\text{LowRisk}$ condition resulted in the actual output $O = \{\text{approved}\}$.

$\text{Bmi}$ is a small BPEL process that computes the Body Mass Index (BMI) from two parameters: height and weight. It invokes one external web service. Based on the value returned by this web service, the process decides if the provided result should be “underweight”, “healthy”, “overweight”, “obese”, or “very obese”.

For our tests, we used the following scenarios. For BMI-V1, we changed (mutated) the $\text{IFUnderWeight}$ condition from $\text{bmiVal} \leq 19$ to $\text{bmiVal} \not= 19$. For input $I = \{\text{weight} = 105, \text{height} = 160\}$, the expected output was $E = \{\text{very obese}\}$, but due to the mutation, the program led to output $O = \{\text{underweight}\}$. Mutating the $\text{IFHealthy}$ condition from $\text{bmiVal} \leq 25$ to $\text{bmiVal} > 25$ yielded the faulty process of scenario BMI-V3. The revealing test case had $I = \{\text{weight} = 75, \text{height} = 160\}$ as input, and $E = \{\text{overweight}\}$ as expected output. Contradicting $E$, the actual output was $O = \{\text{healthy}\}$. For BMI-V4, we changed the $\text{IFUnderWeight}$ condition from $\text{bmiVal} \leq 19$ to $\text{bmiVal} = 19$. The input $I = \{\text{weight} = 50, \text{height} = 160\}$ was expected to yield in output $E = \{\text{healthy}\}$, but the mutated program returned $O = \{\text{underweight}\}$ instead. The same $\text{IFUnderWeight}$ condition was changed to $\text{bmiVal} > 19$ for scenario BMI-V5. The revealing test case had as input $I = \{\text{weight} = 75, \text{height} = 160\}$ with the experienced output $O = \{\text{underweight}\}$ contradicting the expected one $E = \{\text{overweight}\}$.

For the Triangle example as explained in the introduction, we considered the following mutations. For TRI-V1, $\text{condA} \ (a_0 > 0) \wedge (b_0 > 0) \wedge (c_0 > 0)$ was changed to $(a_0 < 0) \wedge (b_0 > 0) \wedge (c_0 > 0)$. The test case with input $I = \{a = 2, b = 2, c = 3\}$ was expected to result in the output $E = \{\text{IS}\}$ (i.e. “isosceles”). The mutated program, however, returned $O = \{\text{NEG}\}$ (i.e. “No triangle”) as output. For TRI-V2, $\text{condA}$ condition was mutated to $(a_0 > 0) \wedge (b_0 < 0) \wedge (c_0 > 0)$. The failing test case used as input $I = \{a = 1, b = 1, c = 1\}$, where the expected output was $E = \{\text{EQ}\}$ (i.e. “equilateral”). Contradicting $E$, the program returned the incorrect output $O = \{\text{NEG}\}$. In TRI-V3, $\text{condA}$ was changed to $(a_0 > 0) \wedge (b_0 > 0) \wedge (c_0 < 0)$, again the revealing test case having $I = \{a = 1, b = 1, c = 1\}$ as input and $E = \{\text{EQ}\}$ as expected output. Like for TRI-V1 and TRI-V2, the actual output was $O = \{\text{NEG}\}$, contradicting $E$. For TRI-V4 the $\text{condC}$ was changed from $(a_0 = b_0) \wedge (b_0 = c_0)$ to $(a_0 = b_0) \vee (b_0 = c_0)$. The revealing test case $t$ had $I = \{a = 2, b = 2, c = 1\}$ as input, and was expected to return $E = \{\text{IS}\}$, but due to the mutation, the program returned $O = \{\text{EQ}\}$.

One can argue that the number of constraints for the examples presented in the paper is rather small. In the SOA domain, however, most of the system’s functionality is wrapped in web services, about which we have (and model) partial information only. The prime purpose of a business process is to integrate different web services in order to achieve a business goal. In particular this means, the focus of a business process is likely on the handling and integration of the involved web services’ in- and outputs.

Table 1 shows the results of our experiments. The total numbers of program’s inputs and outputs are given as $\text{Inputs}$ and $\text{Outputs}$ in the table. $\text{WS}$ refers to the number of external web services invoked by the BPEL process. The number of constraints and variables derived for a path are reported as $\text{CO}$ and $\text{VAR}$, and $\text{Obs}$ gives the number of observations. Column $\text{AllSols}$ reports the number of all possible single fault solutions for a particular example. Similarly, $\text{T min}$ (s) and $\text{T max}$ (s) define the minimum and maximum time in seconds as needed to compute the diagnoses, where $\text{T avg}$ (s) is the average total time over ten runs.

For our results we had some interesting findings. First, the execution time taken by our approach for diagnosing BPEL process is negligible in contrast to the time needed for testing such BPEL examples [6]. Second, the diagnosis approach worked best for the BMI-V1 example, where our diagnoses removed half of the statements from the diagnostic scope. For the other variants, the reduction was about 25%. For the running example “Triangle”, we got a reduction of 40% for three programs, but in TRI-V4, not a single statement could be excluded from the search for the fault.

Figure 4 compares the trace size and the diagnoses size for all the program variants used in our experiments. This figure shows that the amount of components that need to be investigated could be reduced.
Table 1: Single Faults Diagnoses.

<table>
<thead>
<tr>
<th>Program</th>
<th>Inputs</th>
<th>Outputs</th>
<th>WS</th>
<th>S</th>
<th>S'</th>
<th>CO</th>
<th>Obs</th>
<th>Var</th>
<th>AllSols</th>
<th>T min (s)</th>
<th>T max (s)</th>
<th>T avg (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loan-V1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>9</td>
<td>17</td>
<td>3</td>
<td>14</td>
<td>7</td>
<td>0.129246</td>
<td>0.135582</td>
<td>0.1326410</td>
</tr>
<tr>
<td>Loan-V2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>9</td>
<td>17</td>
<td>3</td>
<td>14</td>
<td>7</td>
<td>0.243236</td>
<td>0.302773</td>
<td>0.2548036</td>
</tr>
<tr>
<td>Loan-V4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>9</td>
<td>17</td>
<td>3</td>
<td>14</td>
<td>8</td>
<td>0.023699</td>
<td>0.027242</td>
<td>0.0252019</td>
</tr>
<tr>
<td>Loan-V5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>9</td>
<td>17</td>
<td>3</td>
<td>14</td>
<td>7</td>
<td>0.021730</td>
<td>0.025869</td>
<td>0.0238879</td>
</tr>
<tr>
<td>Bmi-V1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>15</td>
<td>3</td>
<td>11</td>
<td>4</td>
<td>0.008145</td>
<td>0.009979</td>
<td>0.008565</td>
</tr>
<tr>
<td>Bmi-V3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>18</td>
<td>3</td>
<td>13</td>
<td>7</td>
<td>0.008578</td>
<td>0.010395</td>
<td>0.0092091</td>
</tr>
<tr>
<td>Bmi-V4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>15</td>
<td>3</td>
<td>11</td>
<td>6</td>
<td>0.008361</td>
<td>0.010359</td>
<td>0.0091162</td>
</tr>
<tr>
<td>Bmi-V5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>15</td>
<td>3</td>
<td>11</td>
<td>6</td>
<td>0.008381</td>
<td>0.009963</td>
<td>0.0091609</td>
</tr>
<tr>
<td>Tri-V1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>12</td>
<td>4</td>
<td>10</td>
<td>3</td>
<td>0.003844</td>
<td>0.005370</td>
<td>0.0043121</td>
</tr>
<tr>
<td>Tri-V2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>12</td>
<td>4</td>
<td>10</td>
<td>3</td>
<td>0.004226</td>
<td>0.005441</td>
<td>0.0043277</td>
</tr>
<tr>
<td>Tri-V3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>12</td>
<td>4</td>
<td>10</td>
<td>3</td>
<td>0.003918</td>
<td>0.005127</td>
<td>0.0043014</td>
</tr>
<tr>
<td>Tri-V4</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>21</td>
<td>4</td>
<td>21</td>
<td>10</td>
<td>0.004029</td>
<td>0.006739</td>
<td>0.0048355</td>
</tr>
</tbody>
</table>

5 Summary and Future Work

In this paper we propose a focused approach at functional diagnosis of BPEL processes. Implementing a consistency-oriented model-based diagnosis approach, we derive corresponding diagnoses directly from a constraint satisfaction problem that we derive from a BPEL flow graph capturing the flow structure and annotated with any available knowledge. Aiming to accommodate the sheer stunning complexity of such programs, we proposed an approach that focuses on a single execution path (as specified by the input part of the observations), instead of having to consider a full-fledged temporal model like an automaton. The diagnoses then tell which constraints along the path could resolve the issue, or where a correct program might deviate from this path. Specifically in the latter case we do have to take into account which parts of the path are contained in the scenario of a diagnosis, as everything outside the path is left “open”.

While first experiments showed promising performance in terms of run-time, we certainly have to conduct more experiments with further examples. As we did experience memory issues with some of our initial examples (not mentioned in the paper), further experiments are needed in order to come up with a modeling guide for achieving good performance in practice. A very interesting part of our future work will be to explore the effects of weakening and strengthening the auxiliary data (i.e., pre- and post-conditions) on the diagnostic performance. While we currently focus on a functional diagnosis in terms of a weak fault model and persistent faults, strong fault models and the consideration of intermittent faults in the context of reliability issues (and other non-functional properties) that take also dynamic effects into account, will be interesting topics for future research.

Acknowledgment

This work was partially supported by the Austrian Science Fund (FWF): P23313-N23, P22959-N23.

References


