Abstract—In the fierce competition on today’s software market, Service-Oriented Architectures (SOAs) are an established design paradigm. Essential concepts like modularization, reuse, and the corresponding IP core business are inherently supported in the development and operation of SOAs that offer flexibility in many aspects and thus optimal conditions also for heterogeneous system developments. The intrinsics of large and complex SOA enterprises, however, require us to adopt and evolve our verification technology, in order to achieve expected software quality levels. In this paper, we contribute to this challenge by proposing a constraint-based testing approach for SOAs. In our work, we augment a SOA’s BPEL business model with pre- and postcondition contracts defining essential component traits, and derive a suite of feasible test cases to be executed after assessing its quality via corresponding coverage criteria. We illustrate our approach’s viability via a running example as well as experimental results, and discuss current and envisioned automation levels in the context of a test and diagnosis workflow.

I. INTRODUCTION

Considering the hard competition and diversity on the software market, as well as the ever-growing system size, i.e., regarding on-line software and data services, Service-Oriented Architectures (SOAs) with their specific advantages emerged at exactly the right time. With modularization, component reuse, and the related Intellectual Property core business models inherently supported in SOA-based system development, maintenance, and operation, such architectures have been gaining in attention and are nowadays widely accepted in industry [1]. Large, scalable SOA-based systems nowadays orchestrate a multitude of services, registries, brokers, mediators, message buses, et cetera, exploiting, for instance, the offered flexibility behind dynamic, loose couplings, and leveraging research and development, for example, in the contexts of adaptive [2] and self-healing distributed systems [3].

While achieving established, required, and, i.e., requested software quality levels is obviously a prerequisite for successful business cases, this is still a cumbersome task in the case of large, complex SOA-based systems, despite the obvious demand for appropriate technology. Hug and heterogeneous SOA-based software landscapes exacerbate [4], for instance, challenges in the context of controllability, observability, and distribution [5], [6], [7], besides a suitable test process having to accommodate multiple abstraction and description levels associated with the individual components and their models.

In this paper, we contribute to addressing these challenges by proposing a constraint-oriented model-based grey box testing approach for SOAs. As part of our envisioned diagnosis and testing strategy for SOAs depicted in [8], in this paper our focus is on testing service compositions, extending preliminary work presented in [9]. As underlying model driving the testing workflow, we use an augmented version of a SOA’s business process model. That is, we augment an available business process model with pre- and postcondition contracts catching essential details needed for creating an effective and efficient test suite. We start with a business model described in BPEL [10] that orchestrates multiple services/components in order to achieve its goal. BPEL is an XML-based business process executable language, where BPEL processes can be abstract or executable. We are interested in the latter, i.e. the synchronous variant. That is, we handle most basic activities such as receive, reply, assign and invoke, as well as the structured ones if, while and sequence. While specific concurrent scenarios with no (hidden) sub-process interaction can be easily handled by our approach in a sequential way, the flow activity as used for concurrency is part of future research.

After assessing a test suite’s quality with appropriate metrics, the test cases are executed accordingly, using presently available technology. Our focus in this paper will be on the pre- and postconditions used, creating effective test suites, as well as the automation levels currently achieved and envisioned for the future in the context of a SOA testing workflow.

We will illustrate our concept in the context of a running example taken from [11]. Figure 1 depicts the business model of this well-known BankLoan example on an abstract level, while Figure 2 offers a more technical view. The purpose of this process is to handle, i.e., approve or reject, loan requests.

Starting with a client requesting a loan, the process divides requests into two classes. Loan requests below 10,000 credits, and those for amounts starting at 10,000 credits. For the first class, a risk-level is computed immediately, where the corresponding BPEL service calculateRisk takes information like the clientID and the loan amount into account. Low-risk requests then are issued an immediate approval, while high-risk requests demand for a thorough assessment, as it is required also in case of a request from the second class. The relevant service is thoroughAssessment.

While implementation details about these two essential services are unavailable, an abstract model, i.e. defining the interfaces, has to be known for system integration purposes. This is part of the information we aim to attach to the model via pre- and postcondition contracts. For our example, we assume calculateRisk to suggest a low risk for clients with excellent bank records, or whenever the amount is below 1000 credits. This can be easily captured in corresponding pre- and
postconditions (see Section III) attached to the corresponding process component. With illustrations of definitions and ideas in the context of this running example, we aim to make the details of our approach more accessible. We complement this with first experimental results.

Our paper is structured as follows: In Section II we depict our contribution to the challenge of automated testing of SOAs. Section III gives the details of the constraint-based test case generation and execution using augmented BPEL process models, with experimental results reported in Section IV. Covering related work in Section V, we conclude in Section VI depicting also future work.

II. AUTOMATED SOA TESTING

In Figure 3, we illustrate the architecture of our tool BPELTester as well as the links to external tools required for automating the whole test cycle comprising test suite generation and execution. BPELTester takes the SOA definition comprising BPEL and WSDL files and converts them into a control flow graph representation that might be annotated using pre- and postconditions of service invocations. This conversion is done by the AnalysisModule. From the control flow graph representation, the Test Suite Generator derives abstract test cases (paths) that represent particular traversals through the control flow graph. Each such traversal represents a particular execution of the SOA process model. In order to compute corresponding test cases, we use the MINION constraint solver [12] to compute all the necessary inputs and expected outputs that characterize such an execution. The resulting test suite is taken by the Test2Unit module to finally derive the appropriate input for the BPELUnit [13] tool that is used for actually executing the generated tests.

We implemented BPELTester in Java, relying on Eclipse BPEL Designer and the Apache ODE deployment engine [14]. Our tool provides two test execution modes, i.e., a real-time and a simulated testing mode. Both, test case generation as well as their execution, are controlled by BPELTester. Hence, the whole testing cycle is automated, provided that BPEL and WSDL descriptions as well as pre- and postconditions for service invocations are available. In the remaining part of this paper, we outline the test case generation part in detail and also present and discuss the obtained first empirical results for simple samples.

Complementing testing, we plan to add diagnosis functionality to BPELTester. The idea here is to call a diagnosis engine in case a unit test fails. The diagnosis engine then can make use of the constraint representation used for testing explicitly. Moreover, generating new test cases based on obtained diagnosis results, that aim at distinguishing different diagnosis candidates, is possible. For the underlying method and theory we refer the interested reader to [15] where the authors describe the use of constraints obtained from a program’s source code for diagnosis and test case generation. However, an adaptation to the SOA domain considering dynamic service invocations as well as incomplete knowledge about the behavior of the overall system seems to be a necessity.

1http://www.eclipse.org/bpel/
III. Test Case Generation and Execution

As the underlying model driving our test workflow, we use a BPEL Flow Graph derived from a process’ BPEL model augmented with pre- and postcondition contracts.

**Definition 1 (BPEL Flow Graph):** A BPEL Flow Graph $G$ is a tuple $(\{V, E, v_s, F, \Gamma_A, \Gamma_C\})$, where vertices $v \in V$ represent BPEL process activities, $E$ is the set of edges $e = (v_i, v_j)$ which correspond to the connections between BPEL activities, $v_s \in V$ is the start vertex, $F \subseteq V$ is the set of graph $G$’s leaf vertices, and $\Gamma_A$ as well as $\Gamma_C$ are functions that map vertices to activity assignments and conditions respectively.

The flow graph for our running example is quite similar to the graphical representation of the original process in Figure 2. The vertices for this process are defined by the following activities: receiveInput ($v_{\text{IL}}$), AssignLoan ($v_{\text{AL}}$), IfLoan ($v_{\text{IL}}$) and ($v_{\text{EIL}}$), AssignInRisk ($v_{\text{AIR}}$), InvokeRiskService ($v_{\text{IRS}}$), AssignOutRisk ($v_{\text{OAR}}$), IfLowRisk ($v_{\text{ILR}}$) and ($v_{\text{EILR}}$), AssignApproved ($v_{\text{AA}}$), AssignInAssess ($v_{\text{AAA}}$), InvokeAssessRisk ($v_{\text{IAR}}$), AssignOutAssess ($v_{\text{OAA}}$), and replyOutput ($v_{\text{IO}}$).

We store all the data related to a particular activity with the corresponding vertex, i.e. input variables, output variables, assignments, as well as conditions for structured activities if, else if, and while. Essential are also the pre- and post conditions related to an activity. The conditions for vertex $v_{\text{IL}}$ (the receiveInput activity) are defined, for example, as follows:

- **Pre-condition:** $\$\text{input\_loan} > 0$
- **Post-condition:** $\$\text{loan} = \$\text{input\_loan}$

While due to the pre-condition only loan requests with positive amounts are allowed, the post condition ensures that the local variable “loan” is assigned the actual loan amount. For these pre- and postconditions we use the same language as is used for BPEL expressions, i.e., XPATH [16], where we currently support the usual Boolean operations. Naturally, we define pre- and postconditions also for Invoke activities. That is, for the computeRisk service, the behavior suggested in the introduction is defined via the (partial) postcondition ($\$\text{loan} \geq 1000$) ∨ ($\$\text{risk} == 0$).

Obviously, those paths in a BPEL Flow Graph leading from start vertex $v_s$ via some edges in $E$ and intermediate nodes in $V$ to some leaf vertex in $F$ describe the various scenarios covered by the process. For our corresponding functional test case extraction, we define a path $\pi$ formally as follows.

**Definition 2 (Path):** Given a BPEL Flow Graph $G$, a sequence of vertices $\pi = v_1v_2 \ldots v_n$ is a path in $G$ iff (1) for all $i \in \{1, \ldots, n\}$ we have $v_i \in V$, (2) the sequence starts with the start vertex $v_s$, i.e., $v_1 = v_s$, (3) $v_n$ is a leaf vertex, i.e., $v_n \in F$, and (4) for all $i \in \{1, \ldots, n - 1\}$ we have $(v_i, v_{i+1}) \in E$.

In our running example a low-risk low-amount loan request is treated via the following path (Path 1): $\pi = v_{\text{IL}}v_{\text{AL}}v_{\text{AIR}}v_{\text{IRS}}v_{\text{OAR}}v_{\text{IAR}}v_{\text{AAA}}v_{\text{IO}}$.

We will use such paths to derive our test cases. That is, as a first step, a search based algorithm traverses the graph in a depth-first search manner in order to extract the paths from $v_s$ to the leaf vertices. For our running example there are three such possible paths, i.e., the first for low-risk low-amount requests leading to an immediate response approving the request, the second and third paths requiring a more thorough assessment. For generating the test cases, we are particularly interested in the values assigned to BPEL process variables causing the execution of a specific path. As such an assignment might not exist, i.e. for colliding contracts, we have to verify for any path $\pi$ whether there is an actual assignment such that $\pi$ becomes feasible. To this purpose, we define path condition $c(\pi)$ that comprises $\pi$’s vertices’ assignments and conditions.

**Definition 3 (Path Condition):** Given a path $\pi = v_1 \ldots v_n$ of some BPEL Flow Graph $G$, path $\pi$’s path condition $c(\pi)$ is a sequence of assignments and conditionals of $\pi$’s vertices defined as $c(\pi) = \Gamma_A(v_1) \cup \Gamma_C(v_1) \ldots \Gamma_A(v_n) \cup \Gamma_C(v_n)$, where variables are replaced by indexed variables in order to ensure static single assignment form [17].

A static single assignment form (SSA) [17] ensures that every variable is defined only once in a program. For our path condition, we peruse indexed variables (i.e. “temporal” variable instances clocked by assignment events). Starting with index 0, whenever a variable is defined, i.e., occurs at the right side of an assignment, the corresponding index is incremented. Every subsequent reference then uses this new “instance”, unless the index is increased again due to a redefinition of the variable (and so on). For the three paths in our exemplary business process, the corresponding path conditions are given in Figures 4, 5, and 6 respectively.

Expressing a path condition $c(\pi)$ in SSA, we can easily determine $\pi$’s feasibility using any constraint solver. To this purpose, the assignments have to be converted to conditions, i.e., by replacing the assignment operator with the sign of the equality operator and the conditions can be left as they are.

**Definition 4 (Feasible Path):** A path $\pi$ in a BPEL Flow Graph $G$ is feasible if its path condition $c(\pi)$ is satisfiable.

In order to check a path’s conditions’ feasibility we convert them into the input language of the MINION constraint solver [12]. MINION is an out of the box, open source constraint solver, whose syntax requires a little more effort on modeling the constraints than competing implementations, e.g., MINION does not support different operators to be used...
For actually executing the derived test cases, we rely on the BPEL Unit testing tool [13]. By specifying inputs and expected outputs via a GUI it lets a user generate an XML test suite, where we derive an appropriate representation automatically from the generated test cases. The conversion algorithm traverses all feasible paths and converts each receive activity into a send activity, and each reply activity into a corresponding receive condition as required by the BPEL Unit Test Suite. Furthermore, we add for any invoke activity the corresponding “partner track” information to the XML test suite. An example test case for Path 3 is given below.

```xml
<tes:data>
  <ban:LoanRequest>
    <ban:ClientId>20</ban:ClientId>
    <ban:LoanAmount>10001</ban:LoanAmount>
  </ban:LoanRequest>
</tes:data>
<tes:send>
  <tes:receive fault="false">
      <tes:condition>
          <tes:expression>//ban:LoanResponse</tes:expression>
          <tes:value>'AssessRiskPLResponse'</tes:value>
      </tes:condition>
      <tes:receive>
          ...
      <tes:partnerTrack name="AssessPartner">
```

**IV. EMPIRICAL RESULTS**

In this section we report first empirical results obtained using the Java implementation of our BPEL Tester tool. We considered three examples in our evaluation: LoanApproval described previously in this paper, ATM², and a simple hand-

²http://docs.jboss.com/jboss/bpel/v1.1/userguide/tutorial.atm.html
TABLE I
EXAMPLES

<table>
<thead>
<tr>
<th>Prog</th>
<th>BPEL Activities Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loan</td>
<td>Receive, Reply, Assign, If, Else, Invoke, Sequence</td>
</tr>
<tr>
<td>ATM</td>
<td>Receive, Reply, Assign, If, Else if, While, Invoke, Sequence</td>
</tr>
<tr>
<td>While</td>
<td>Receive, Reply, Assign, While, Sequence</td>
</tr>
</tbody>
</table>

crafted example comprising a while statement within the process definition. From the first two examples we removed flow and fault handling BPEL constructs currently not supported by BPELTester. Table I list the activities used in these examples.

In Table II we summarize the obtained test generation results, i.e., the number of BPEL activities in any process $n$, the number of paths $p$, the maximum path length $mLen$ varying from 10 to 50 (in case of examples with while loops), the minimum and maximum path lengths of the BPEL process $\text{minP}$ and $\text{maxP}$, and the minimum and maximum numbers of MINION constraints $\text{minC}$ and $\text{maxC}$. $\text{totalT}$ represents the total time in milliseconds it took to generate the executable test cases. The time for checking the paths’ feasibility via constraint solving was always very small ranging from 11 to 26 milliseconds and is thus omitted in the table.

Cardoso [18] explained the complexity of BPEL processes using the control flow complexity (CFC) metric. The test execution tool described in [19] supports test coverage metrics like activity, branch, link and handler coverage. Since having a large number of tests is undesirable for testing web services due to the related costs [20], we investigated our test suites’ quality with respect to coverage via the tool described in [19]. For all considered examples, we attained 100% activity and branch coverage by just the minimum set of paths. In particular, activity and branch coverage for the Loan and While examples reach 100% for the smallest path length with only 3 paths. For the more complex ATM example, we obtain 100% coverage for a minimum path length of 19 with 13 generated paths (see Table II). The coverage progression as a function of the path length is given for the ATM example in Figure IV.

The obtained empirical results are promising and indicate the usefulness of our approach. Even for smaller path lengths we obtained a coverage of 100%, where it took less than 1 second for computing the test suite. However, in order to strengthen the empirical evaluation, we plan to increase the example set by including other larger real-world business processes. Moreover, we also will use different coverage metrics like MC/DC and also the mutation score for evaluating the obtained test suite. It is worth noting that executing the tests took twice the time for generating the test suite.

V. RELATED WORK

Exploiting constraints for software testing is an attractive concept. Gotlieb et al. [21] extracted test cases from programs via a constraint representation of its source code. Whereas our work is quite close to this in principle, the application domain and constraint extraction process are different. In our case we exploit also a component’s pre and post conditions, because, as a matter of fact, a SOA’s services’ actual implementations are hardly available. In this respect we differ also from [22], where, in contrast to our BPEL flow graph, Bentakouk et al. translate a BPEL model into a symbolic transition system (STS) used to extract test cases from. They issue specific warnings for situations where their approach is incomplete due to time-out violations in the construction of the STS. Similar work was presented in [23].

In the context of web service testing, literature reports on mainly three model based testing techniques, i.e., symbolic execution, petri nets, and model checking [24]. Yuan et al. [25] presented a graph search based test case generation of BPEL processes that exploits matrix transformations of control flow graphs, path coverage, and a node classification depending on incoming and outgoing edges. While their approach is close, we differ in the use of pre- and postconditions added to the test paths, aiming to solve the test oracle problem. A slight difference is also in the use of the MINION constraint solver [12] rather than Lp.

The underlying idea behind [26], which relies on an extended Control Flow Graph (XCFG), is to extract all sequential paths from the XCFG, and to combine them into concurrent test paths. Constraints are then collected from these concurrent test paths via backward substitution. In contrast, we transform each sequential path directly into a set of constraints, each set independently checked for satisfiability. For unsatisfiable constraints, the corresponding path is discarded, satisfiable ones produce the corresponding variables used to execute the path. These values define a test case to be included in the test suite under construction.

Also model checking [24] can be exploited in the context of web service testing. For this, BPEL specifications are converted into a formal modeling language like PROMELA [27]. Defining test criteria as formal properties, in a language such as LTL [28], a model checker can be used to search for violations of the properties by the BPEL model. The actual test cases then are derived from the counter examples provided by the model checker for such violations. Zhen et al. [29] applied the same idea to web services and BPEL processes, with an
enhanced version [30] addressing the state space explosion problem inherent with model checking. Moreover, they also developed a tool for the generation of JUnit test cases for automated test execution.

Model based testing techniques using Petri Nets have also been explored extensively. Petri Nets are attractive for modeling concurrent processes and their synchronization, and can be categorized into Plain Petri Nets [31], Colored Petri Nets [32] and High-level Petri Nets. Dong [33] developed a tool for test case generation of BPEL processes using High-level Petri Nets. The basic approach is to build a reachability graph from which test cases can be extracted. The approach has a very high space complexity.

Contract based testing techniques are also relevant to our work. This idea [34] has been applied in WSDL Web Service Technology [35], where the authors argue that the contracts applied at the model level are useful in the automated generation of test oracles, but can be very costly to implement. Also, the assertions are easier to apply in OWL WS technology but are difficult to implement in a WSDL based process model [24]. Dai et al. [36] combine this approach with Petri Nets. They specify contracts using a OWL-S model and transform them into Petri Nets. The test cases are generated based on a Petri Net behavioral analysis. In contrast, we combine contract based testing with symbolic execution in our approach.

For test case execution, we use [37], where we convert the abstract test cases manually to executable ones accepted by the BPELUnit tool [13]. This tool supports simulated as well as real-life testing, accommodating many BPEL engines like Active VOS [38], Oracle BPEL Process Manager[39], and Apache ODE [14]. For simulated testing, a BPEL process is not deployed, rather the intended engine is called through a debug API. In real-life testing mode, a business process is actually deployed on the selected engine and the partner web services are tested using mocks.

The survey of Zakaria et al. [40] gives a very good comparison of different unit testing approaches applied to BPEL processes. One key issue pointed out there is the lack of an empirical evaluation. Surprisingly, only 1 out of 27 considered studies provides results on real-life BPEL processes.

VI. CONCLUSIONS

In this paper, we depicted our BPELTester tool for functional testing of SOA business processes. Our underlying engine derives from a process’s BPEL model a control flow graph, which is traversed to collect the symbolic executions (paths) of the process. Exploiting also augmented data, i.e. contract information like pre- and postconditions suggested by external sources, we determine the feasibility of a symbolic execution via a constraint description of the corresponding assignments and conditions. A representative for a feasible path in the form of a satisfying assignment defines a new test case. BPELTester is written in Java and relies on [13] for the actual test execution. We further use the tool described in [19] for assessing the quality of the obtained test suite.

While first empirical results for smaller samples are promising, for large samples we expect that reaching a 100% coverage becomes infeasible, i.e. in the context of a given budget and associated resource limits. Thus future work will have to identify effective means for focusing the individual tests

<table>
<thead>
<tr>
<th>Prog</th>
<th>n</th>
<th>mL</th>
<th>p</th>
<th>min</th>
<th>max</th>
<th>minC</th>
<th>maxC</th>
<th>totalT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loan</td>
<td>16</td>
<td>10</td>
<td>3</td>
<td>8</td>
<td>10</td>
<td>8</td>
<td>11</td>
<td>160</td>
</tr>
<tr>
<td>ATM</td>
<td>27</td>
<td>10</td>
<td>1</td>
<td>9</td>
<td>12</td>
<td>12</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1</td>
<td>9</td>
<td>12</td>
<td>12</td>
<td>57</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>2</td>
<td>13</td>
<td>12</td>
<td>17</td>
<td>132</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>3</td>
<td>15</td>
<td>12</td>
<td>20</td>
<td>152</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>5</td>
<td>15</td>
<td>12</td>
<td>20</td>
<td>322</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>5</td>
<td>15</td>
<td>12</td>
<td>20</td>
<td>322</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>5</td>
<td>15</td>
<td>12</td>
<td>20</td>
<td>322</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>6</td>
<td>18</td>
<td>12</td>
<td>24</td>
<td>395</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>8</td>
<td>19</td>
<td>12</td>
<td>27</td>
<td>528</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>13</td>
<td>20</td>
<td>12</td>
<td>28</td>
<td>878</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>17</td>
<td>20</td>
<td>12</td>
<td>28</td>
<td>1,151</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>20</td>
<td>20</td>
<td>12</td>
<td>33</td>
<td>9,150</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>1,367</td>
<td>9</td>
<td>40</td>
<td>10</td>
<td>46</td>
<td>108,983</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>12,950</td>
<td>9</td>
<td>50</td>
<td>10</td>
<td>57</td>
<td>1,372,059</td>
<td></td>
</tr>
<tr>
<td>While</td>
<td>8</td>
<td>10</td>
<td>8</td>
<td>10</td>
<td>13</td>
<td>16</td>
<td>169</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>28</td>
<td>8</td>
<td>20</td>
<td>13</td>
<td>28</td>
<td>2,028</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>78</td>
<td>8</td>
<td>30</td>
<td>13</td>
<td>45</td>
<td>6,194</td>
<td></td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>153</td>
<td>8</td>
<td>40</td>
<td>13</td>
<td>62</td>
<td>13,703</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>253</td>
<td>8</td>
<td>50</td>
<td>13</td>
<td>71</td>
<td>23,849</td>
<td></td>
</tr>
</tbody>
</table>

TABLE II
EMPIRICAL RESULTS OBTAINED
in a suite while maintaining a high quality for the associated overall assessment. In this respect, we aim also to include further coverage metrics and will have to identify those most efficient and representative in our context.

For failing test cases, we plan to integrate a diagnostic reasoning engine. On one hand this will be aimed to identify the corresponding faults in a SOA-based system, possibly suggesting further test cases providing the data for discriminating between competing hypotheses. On the other hand also the specification could be flawed, i.e. augmented information regarding pre- and postcondition contracts could introduce flaws that a diagnostic reasoning could help to identify.

Further research will also aim at non-functional SOA traits in the context of its service level agreements (SLAs). Using constraints seems to be a promising strategy whose potential was not sufficiently explored to the best of our knowledge.

ACKNOWLEDGMENT

We would like to thank Schahram Dustdar, Philipp Leitner, and Stefan Schulte for fruitful discussions on the work presented in this paper. This work was partially supported by the Austrian Science Fund (FWF): P23313-N23, P22959-N23.

REFERENCES