A Short Survey on Using Software Error Localization for Service Compositions

Julia Krämer(✉) and Heike Wehrheim

Department of Computer Science, Paderborn University, Paderborn, Germany
{juliadk,wehrheim}@mail.uni-paderborn.de

Abstract. In modern software development, paradigms like component-based software engineering (CBSE) and service-oriented architectures (SOA) emphasize the construction of large software systems out of existing components or services. Therein, a *service* is a self-contained piece of software, which adheres to a specified *interface*. In a model-based software design, this interface constitutes our sole knowledge of the service at design time, while service implementations are not available. Therefore, correctness checks or detection of potential errors in service compositions has to be carried out without the possibility of *executing* services. This challenges the usage of standard software error localization techniques for service compositions. In this paper, we review state-of-the-art approaches for error localization of software and discuss their applicability to service compositions.

1 Introduction

Debugging, i.e., the *detection*, *localization* and *correction* of software errors, is one of the most time-intensive tasks in software development. Within this process, error localization is considered the most expensive task [43]. In order to support developers in debugging, a lot of research effort has been spent on the deployment of *automated* error localization methods. Today, existing error localization methods for software are numerous. They can broadly be separated into two categories.

(1) *Approaches based on the inspection of test cases.* In this category, we find error localization methods like *delta debugging* introduced by Zeller [17,49–51] or the approaches underlying the tools *Tarantula* [27], *Pinpoint* [14] or *AMPLE* [19].
(2) *Approaches based on the computation of dependence information between program statements.* Herein, we locate all techniques based on *static program slicing* as originally introduced by Weiser [46] as well as dynamic slicing.

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

M. Aiello et al. (Eds.): ESOCC 2016, LNCS 9846, pp. 248–262, 2016.
DOI: 10.1007/978-3-319-44482-6_16

When it comes to a model-driven design approach of service composition (or to model-driven software development in general), the situation is different. On the one hand, a model typically abstracts from details of the final software, thus facilitating the construction of automatic methods and tools for error detection (like being done in numerous settings, for functional as well as QoS requirements, e.g. [12,21,39,42]). On the other hand, the *localization* of errors, once correctness checks have reported it, lacks automated methods and tool support. So far, to the best of our knowledge, automated, tool-based approaches for localizing faults in models of service compositions do not exist, at least when it comes to *functional correctness*, i.e., the adherence of the model to functional requirements. With respect to performance analysis of systems, feasible approaches to localize components that negatively impact the overall performance of the system, have been devised in the area of *performance blame analysis* [13,18].

Unfortunately, this lack in tool support cannot easily be amended by applying the abundant existing approaches for standard software development to the service composition approach. The reason is rooted in fact that almost all existing approaches in the standard software setting rely more or less on the availability of execution traces, both faulty and correct, or even the possibility to execute the programs under consideration at will. While this requirement is entirely unproblematic in the software setting, for service composition it is a veritable obstacle, as services, which are offered by external providers and possibly charged for their use, may not be available for execution during design time and fault analysis.

Contribution. In this paper, we survey existing error localization techniques for software, analyze their applicability to *models of service compositions* and propose suitable adaptions. Our focus is on *functional* correctness, more specifically, the adherence of the service composition to specified pre- and postconditions. We assume that services are solely specified in terms of their pre- and postconditions (more precisely, their *interface* specification) and that no other information is available about services. In particular, no implementation is given and thus, they cannot be arbitrarily executed. In this setting, error localization can be rephrased as the task of locating the precise service call, which is responsible for the service composition to invalidate the postcondition when started in a state satisfying the precondition.

In comparison to existing surveys, such as [47] and [3], which focus on methods of the first category, we also investigate methods of the second category and thus, include novel methods for error localization, especially formula-based approaches such as [28–30] and [32]. In contrast to [47] and [3], we do not only review existing methods but also examine their applicability to service compositions.

Organization of the Paper. We introduce basic terminology (services and service compositions) in Sect. 2. In Sect. 3, we present the most important error localization methods for software and discuss their usability for service compositions in the context of model-based software design. We conclude the paper with a conclusion and future work in Sect. 4.

2 Services and Service Compositions

Services, i.e. self-contained software components, which can be used platform independent, are at the core of Service-Oriented Architectures (SOA). In this section, we introduce service descriptions, which constitute all information about a service, and service composition as depicted in Fig. 1. We denote service composition in a textual representation inspired by *service effect specifications* (SEFFs) of [11] (making some of the notations closer to programming languages), while we still use standard concepts of workflow modelling like sequential composition, decisions and repetition. Possible alternative representations for service compositions include graphical or structural notations for workflow modelling like WS-BPEL [40]. The following definition specifies our textual representation of services formally.

Definition 1. *Let Serv be a set of given services, Types be a set of types and Var be a set of variables. The set of all* service compositions $\mathcal{SC}$ *is given by the following grammar in Backus-Naur-form:*

$$\mathcal{SC} \ni \varphi, \psi ::= Skip \mid \varphi; \psi \mid \mathsf{if}\ \mathsf{B}\ \mathsf{then}\ \varphi\ \mathsf{else}\ \psi \mid \mathsf{while}\ \mathsf{B}\ \mathsf{do}\ \varphi \mid T\ x = \mathsf{S}(x_1, \ldots, x_n) \mid \mathsf{foreach}\ x\ \mathsf{in}\ Set\ \mathsf{do}\ \varphi,$$

where $x, x_1, \ldots, x_n \in Var$, $S \in Serv$, $T \in Types$ *and Set is a set.* B *is a predicate in propositional logic with the logical constants* true, false *and service calls* S *as atomic formulas.*

Please note that we use assignments in Fig. 1, which are not service calls, for example, in Line 1, Line 3 and Line 8. We consider these assignment as very basic service calls usually not offered by an external provider but by the service specification language. Thus, we do not write them down as service call.

The service composition GVRes in Fig. 1 contains the service restaurantIn that retrieves all restaurants near a given location, the service isVegan, that tests whether a given restaurant offers vegan food, the service validate that provides the rating of a restaurant, and finally, the service isGoodRating, which specifies when a rating is considered a *good* rating. The purpose of the service composition GVRes is to compute the set B of all vegan restaurants with a good rating near a specific location L provided by the user. However, it is faulty. While the purpose of the `foreach`-loop is to filter all the restaurants with a good rating, the negation in the second `if`-statement (Line 7) causes only bad restaurants to be in the set B. At the best, fault localization would precisely indicate the condition of the `if`-statement `!(IsGoodRating(y))` as the location of the error.

The semantics of single services is cruel to the correctness of a service composition. We specify the semantics using *service descriptions*, which include input and output variables as well as pre- and postconditions (or effects, all together typically called IOPE, like in WSDL[1]).

[1] https://www.w3.org/TR/wsdl.

```
1  Location L:=input();
2  Set<Restaurant> A:=
       restaurantsIn(L);
3  Set<Restaurant> B:=emptyset;
4  foreach z in A do
5    if isVegan(z) then
6      Rating y:=validate(z);
7      if !(isGoodRating(y)) then
8        B:=B union {z};
9      else
10       Skip
11   else
12     Skip
13 return B;
```

Fig. 1. The service composition GVRes

Definition 2. *A* service description SD *is a tuple* $\mathsf{SD} = (\mathsf{I}, \mathsf{O}, \mathsf{Pre}, \mathsf{Post})$ *such that*

- I *and* O *are disjoint sets of input and output variables,*
- Pre *and* Post *are first-order logic formulas, which describe the precondition and the effect (postcondition) of the service, respectively.*

All free variables (i.e. all variables not bound by a quantifier) in Pre *are elements of* I *and all free variables in* Post *are elements of* $\mathsf{I} \cup \mathsf{O}$.

The service validate has the input variable z of type Restaurant and the output variable y of type Rating. Its postcondition guarantees that the returned rating is indeed a rating for the given restaurant if the input is indeed a vegan restaurant.

Service compositions are also specified using service description, e.g. the service composition GVRes has the input variable L of type Location, the output variable B of type Set<Restaurant>, the pre- and postcondition

$$\mathsf{Pre}_{\mathsf{GVRes}} = \mathsf{true}$$
$$\mathsf{Post}_{\mathsf{GVRes}} = \forall b \in B : \mathsf{isVegan}(\mathsf{b}) \wedge \mathsf{isGoodRating}(\mathsf{validate}(\mathsf{b})).$$

In the following, we say that a service composition is *functionally correct* with respect to a precondition Pre and a postcondition Post, if we can prove that for each input to a service composition, which satisfies the precondition, the output satisfies the postcondition. We say, that a service composition *contains an error*, if it is not functionally correct. The service composition in Fig. 1 will thus be functionally correct if it ensures that no bad vegan or non-vegan restaurant is returned (which is not the case). It can be proven that a service composition is or is not functionally correct, for example, using the approach in [44].

3 Survey on Error Localization

So far, error localization in service compositions has been a sparsely researched topic and only few approaches are known. In contrast, many localization methods for standard software (especially for imperative program) are known. Unfortunately, while imperative programs and service compositions are syntactically similar, they differ in their nature. While error location methods for programs can usually safely assume that the whole program can be executed arbitrarily, this is not the case for service compositions. At the time of analysis, the services called in the composition are in general not available for execution. The reason is that services are usually not locally available, but offered by external providers and charged for their usage. Thus, depending on the concrete services, their repeated execution for testing purposes might either not be given at the moment of analysis, or be economically infeasible.

Thus, in order to make use of the rich source of error localization methods for standard software for service composition, we need to investigate how these methods can be adapted – if at all.

Remark 1. The services and service compositions we discuss cannot be compared to *dynamic* web services, in the sense of applications written in PHP or JavaScript involving dynamically generated web pages or client-server-interaction. Therefore, our setting is very different from the setting in [6–9,38,45] and thus, these approaches are inapplicable in our setting.

In this section, we first establish a set of criteria to evaluate existing automated error localization methods. Subsequently, we present an overview on existing error localization methods of both categories when applied to service compositions as in Definition 1 instead of to software.

3.1 Criteria for Error Localization Approaches

In [3,47], criteria to evaluate error localization methods for software are discussed. We use a subset of these criteria, slightly adapted to the special challenges arising in the context of service compositions (Fig 2).

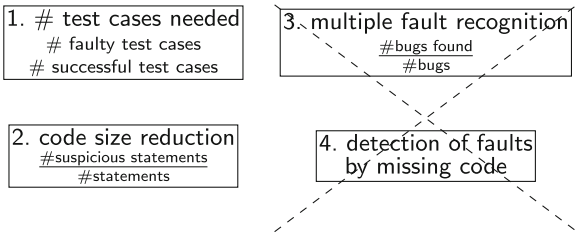

Fig. 2. Criteria for error localization methods and our choice

Number of test cases needed: In model-driven software design, one cannot execute services at *design* time. In the best case, few test cases are available in form of input/output pairs witnessing erroneous behavior, for example resulting from a previous model checking analysis.
Our first evaluation criteria is thus the number of (faulty /correct) test cases a technique needs.

Code size reduction: The second criteria we use is code size reduction, i.e., the percentage of suspicious statements (in which the fault is potentially located) returned by the error localization method with respect to all statements.

Another criterion, which is often used is multiple fault recognition, i.e. the possibility of discovering multiple bugs at once. We do not use it here since service compositions tend to be relatively small, and verification and error localization can thus be executed several times to find several bugs. Detection of faults caused by missing code is not a criterion of primary interest, as the results of existing approaches in general fail w.r.t. multiple bug detection to be specific enough to be of use in a setting where the programs to be analyzed consist of only few lines of code.

3.2 Error Localization Approaches in Service Compositions

In the following, we discuss different error localization methods for standard software. We group approaches, which are similar w.r.t. the number of tests cases they need to be applicable. If necessary, we further distinguish methods by their overall approach, for example, whether it relies on statistics or not.

Neither Relying on Test Cases Nor on Execution. We start our survey with *static slicing*, which also was the first error localization method proposed in 1981 by Weiser [46]. Slicing in general means to cut out statements, which cannot influence a certain variable or a certain property. The "influence" is captured by a number of dependency relations between program statements, e.g., a statement within a branch of a decision depends on the condition of the decision. With respect to error localization this means that the number of statements possibly responsible for the error can be reduced by slicing. Slicing approaches can mainly be divided into *static* and *dynamic* slicing. Whereas the first can be obtained without executing the program and thus, does also not rely on any tests, dynamic slicing gathers information during execution. In [52], it is stated that a static slice definitely contains the bug if it is contained in a Boolean condition or an assignment. Unfortunately, static slices are the largest ones among all slices. Nevertheless, static slicing can easily be modified to be used on service compositions, for example, in [37], static slicing is discussed for software relying on web services.

Application to Service Compositions. For faulty service compositions, we compute slices with respect to the intended postcondition. The static slice with respect to our postcondition $\mathsf{Post}_{\mathsf{GVRes}}$ of the service composition in Fig. 1 contains all lines except the lines 9 to 12 (which are uninteresting anyway). We see

that the gain in this case is close to zero. For finding the error, we still need to inspect the entire service composition. This is an effect, which occurs very often in service compositions because data is passed from one service call to the next, and thus service calls often depend on all prior calls. □

Relying on One Faulty Input. All of the following error localization methods need at least one faulty input, i.e., one input, which itself satisfies the precondition, but leads to an output, which does not satisfy the postcondition.

Dynamic Slicing. Dynamic slicing was originally introduced in 1988 in [31]. The key idea to dynamic slicing is to collect all relevant information directly during the execution of the program. In the literature, there are mainly three types of dynamic slices: *data*, *full* and *relevant* slices. They differ in the way they take dependencies between program statements into account: data slices just use data dependencies, full slices also control dependencies, and relevant slices in addition partially include static dependencies, i.e., dependencies on program paths, which are not included in the current dynamic execution, but might be if the control-flow is altered. At first, dynamic slicing was considered *not useful* for error localization [4,5]. In 2005, an experimental evaluation in [52] showed that relevant slices are smaller than static slices, but contained all bugs in the experiments performed on the Siemens test suite [25].

Application to Service Compositions. For service compositions, an *abstract symbolic execution* – i.e., an execution, which does not rely on concrete but on symbolic values for variables – could allow us to use dynamic slicing for error localization. Important questions to be investigated are then whether dynamic slices relying on a symbolic execution are smaller than static slices, and whether all faults are covered. For our example, a symbolic execution would – like for the static slice – return the whole service composition except the lines 9 to 12. We conjecture that this will very often be the case due to the tight dependencies between service calls. □

Trace Formula Approaches. In this section, we consider all approaches to error localization, which basically rely on a trace formula. The original idea to use trace formulas for *verification* was introduced in [16]. The basic idea therein is to code executions of a program (or even whole programs) as logical formulas, employing either propositional or predicate logic. In [44], this basic principle has been used for the verification of service compositions. We mainly consider the error localization approach presented in [48], where a trace formula is encoded as constraint satisfaction problem. In more detail, in [48], a test defining inputs and expected outputs together with its symbolic execution trace, is transformed into a constraint satisfaction problem and solved using an existing constraint solver. The solution to the constraint satisfaction problem allows to easily extract a set of suspicious statements, which can be returned to the user.

In [28,29], a similar approach using partial MaxSMT to locate errors in programs has been implemented in the tool BugAssist. MaxSMT is the maximal

satisfiability problem, which determines the maximal number of clauses in a logical formula that can be simultaneously made true. MaxSMT instances allow to tag clauses as hard (definitely needs to be true) or soft (candidate for not making it true). With respect to error localization, this allows us to state where the error potential is (or definitely not is) by making this a soft (hard) clause. The test input and the property to be verified (e.g., the postcondition) are encoded as hard clauses, whereas the trace formula representing the program is encoded as soft clause. Using partial MaxSMT, a set of clauses is returned, which can simultaneously be set to true. The complement of this set then serves as set of suspicious statements.

Application to Service Compositions. Although we cannot rely on concrete input and outputs for service compositions, it seems worthwhile to investigate whether the approach can be adapted to work with pre- and postconditions instead of test cases. A verification technique like [44] could for instance be used to generate *abstract inputs* leading to errors. Abstract input means that we do not have concrete values but just names for values. Given that this is possible, we could for instance get an abstract input like *city* for L with the following properties (also given via freely chosen names[2]):

$$\begin{aligned} \mathsf{restaurantsIn}(city) &= \{res\} \\ \mathsf{isVegan}(res) & \\ rat &= \mathsf{validate}(res) \\ \neg\mathsf{isGoodRating}(rat) & \end{aligned}$$

Given such a "test case", the trace formula of the given service composition encoded for MaxSMT may look like this:

$$\begin{array}{lr} \underline{\mathsf{L} = city} & \texttt{input} \\ \wedge \underline{A = \{res\}} \wedge B_0 = \emptyset & \texttt{before loop} \\ \wedge \mathsf{isVegan}(res) \wedge y = rat \wedge \neg\mathsf{isGoodRating}(rat) \wedge B_1 = B_0 \cup \{res\} & \texttt{loop once} \\ \wedge \underline{\forall b \in B_1 : \mathsf{isVegan}(b) \wedge \mathsf{isGoodRating}(\mathsf{validate}(b))} & \texttt{postcond.} \end{array}$$

In this example, the underlined clauses are hard clauses, all other clauses are soft. This formula encodes a path through the service composition when "run" on the test case plus the desired postcondition at the end. In order to encode the same trace and the same expected outputs as constraint satisfaction problem (similar to the approach in [48]), we introduce a predicate AB_i per statement i, which represents whether the statement i is *abnormal*. Abnormal statements are candidates for the root cause of the error. For instance, the first statement of the service compositions is then encoded as

$$(\neg AB_1) \Rightarrow A = \{res\}.$$

[2] The SMT solver underlying the verification technique in [44] treats all service calls and types as undefined function symbols, and thus returns just some randomly chosen identifier for instance of these symbols.

Inputs, the precondition and the postcondition are encoded as so-called observations. The encoding of the statements as well as the observations are then given to a constraint solver, which computes valuations for the predicates AB_i.

Both the MaxSMT and the constraint satisfaction encoding lead to a candidate root cause at line 7, which is exactly where the fault is located. □

Another formula-based approach are error invariants [22]. Intuitively, an error invariant is a formula φ at a statement st such that the program input and the trace formula constructed from the beginning to st imply φ, and φ and the trace formula from st to the end of the execution does imply false. *Inductive error invariants*, i.e. error invariants, which hold for several consecutive statements, allow to identify irrelevant transitions in error traces. Afterwards, they are used as an approach similar to [28,29,32] to compute a set of suspicious statements.

Application to Service Compositions. A first idea for using error invariants for error localization in service compositions is to split abstract symbolic error traces at every service call, use the precondition of the service as assertion to be proven to hold after the split, and the postcondition of the service as additional initial assumption for the next part. This allows to analyze service calls one by one. Nevertheless, a lot of solver calls are necessary to analyze all parts of a service composition this way, and therefore experimental studies need to examine the performance of such an approach. □

An extension of error invariants in order to make fault localization flow sensitive is done in [15]. *Flow-sensitive* trace formulas are used to compute suspicious statements with the help of a software model checker and an interpolating theorem prover. In [32], a *full flow-sensitive trace formula* is published, which is again analyzed using partial MaxSMT. Clauses of the trace formula, which belong to the control flow are marked as hard and all others are marked as soft. The *push & pop mechanism* of the solver Yices [20] yields an efficient solution, which gives quite the same code size reduction as BugAssist but is faster. As flow-sensitive and standard trace formulas are very similar, we think that these approaches are also applicable to service compositions.

Relying on One Faulty and One Correct Input. Delta debugging [49–51] is a divide-and-conquer algorithm to compute the smallest difference between a working and a failing test. In [49], delta debugging is applied to changes introduced between the last correct version of a program and the current faulty version. Intuitively, the algorithm splits all existing changes (if it is not only one) into two non-empty subsets and tests, which changes lead to a successful and which changes to an unsuccessful run of the program. Subsequently, the algorithm recursively computes the faulty change in the set of changes that lead to the error. In [51], a very similar strategy is applied to turn test cases into minimal ones, in [50], the delta debugging approach is applied to program states in order to compute the minimal difference between a working and a failing program. Since we typically do not have different correct and faulty variants of a service composition available, this technique seems less applicable to service compositions.

Relying on Several Faulty Inputs. In [30], all faulty inputs and the respective execution traces are encoded into an instance of SAT. The results are used to compute new right-hand sides to assignments in order to correct the program. In service compositions, the right-hand side of assignments are usually service calls, which cannot be modified, just completely replaced. In addition, the methods perform better if there are several faulty inputs, which we typically cannot provide in our setting.

Relying on Several Faulty and Correct Inputs. In this section, we discuss existing error localization approaches, which use several faulty ad several correct tests in order to generate a set of suspicious statements. For a detailed overview on these error localization methods, we refer the interested reader to [3].

Spectrum-Based and Statistical Methods. Tarantula [26,27] is a spectrum-based error localization method, which computes the suspiciousness of a statement by comparing the number of successful and failing test cases, in which the statement has been executed. Different methods to compute the suspiciousness of a statement, for example, using the Jaccard or Ochiai distance are discussed in [1,2]. Statistical methods such as [14,19,33–36] also rely on successful and failing test cases, but compute the suspiciousness with statistical methods. For example, Pinpoint [14] uses data mining methods to correlate successes and faults to determine the most likely faulty component. As we neither have tests nor the implementation of services and thus, cannot rely on multiple faulty and correct test inputs, we do not consider those error localization methods as easily applicable to service compositions.

Set-based Methods. Two very simple and common techniques to error localization are introduced in [41] and compared to more effective methods like the cause transition approach in [17] and the Tarantula approach [26,27] in [52]. The *set-union technique* computes a set of suspicious statements by removing all statements, which are executed by all passing tests, from the set of statements, which are contained in at least one failed test case. In contrast, the *set-intersection technique* computes a set of suspicious statements by removing all statements, which are executed in a single failing test case, from all statements, which are executed by every passed test case. As their effectiveness is already very limited on programs, we do not expect them to perform well in service compositions, especially as we do not have successful test cases at hand.

Relying on Model Checking. In [10], correct traces produced by a model checker are used to localize the error in existing error traces, more specifically, to report *one single* error trace per error, and to generate multiple error traces for multiple faults. The core of their method is to find transitions in error traces, which do not occur in any correct execution. With respect to service compositions, it could be worthwhile to examine whether there exists services, which do

not occur in a correct execution and then, to add the respective service to the set of suspicious ones.

In [24], a SAT-based approach relying on CBMC [16] to minimize counterexamples of model checkers is published. In [23], the difference (in terms of statements) between a correct and a wrong execution is computed and returned to the user as set of suspicious statements. The approach in [23] only relies on a counterexample and then generates program inputs, which do not violate the specification. Again, we consider it worthwhile to investigate, whether the approach can be adapted to work with service compositions.

Remark 2. In general, one distinguishes between *control-* and *data-flow* errors. A control flow error, is an error, which can be corrected by changing the predicate of a branch or a loop.

As the control-flow of models of service compositions and of standard software do not widely differ and as our example shows, applying standard error localization methods to find control-flow errors in service compositions seems promising.

A data-flow error is an incorrect variable state, which occurs during execution and is caused by wrong assignments. In service compositions, variables are only used to pass data from one service call to another service call. Therefore, the root cause of the data-flow error is likely the service call prior to the failing call. We thus think that the *correction* of data-flow errors is more promising to investigate than simply finding data-flow errors.

	ACSR	test cases	app.	category
MaxSMT Approach	8%	one faulty	✓	2
Constraint Satisfaction Approach	—	one faulty	✓	2
Fully Flow-Sensitive TF	11%	one faulty	✓	2
Static Slicing	≈ 30%	—	✓	2
Error Invariants	—	one faulty	(✓)	2
Dynamic Slicing	≈ 30%	execution	(✓)	2
Set Union	1% yield 10% or less	faulty & correct	f	1
Set Intersection	5.5% yield 10% or less	faulty & correct	f	1
Delta Debugging with Cause Transitions (relevant)	35.66% yield 10% or less	faulty & correct	f	1

Fig. 3. Overview on properties of the presented error localization methods. Column *ACSR* shows the Average Code Size Reduction as stated by the respective authors of the approaches, the column *test cases* states the number and kind of test cases needed, or if even executable code is required. In column *app.*, we summarize the applicability of the approach for service compositions. Column "category" refers to the category, to which the approach belongs with respect to our classification in Sect. 1. Note that early works give the code size reduction in "percentage of programs yielding percentage of code size reduction".

4 Conclusion and Future Work

In this paper, we have shown that error localization methods for standard software do not carry over to service compositions easily. Especially, the unavailability or at least the lack of test cases as well as the impossibility to execute service compositions at will, render most error localization methods inapplicable.

Figure 3 summarizes our findings. It seems that, in general, approaches in the second category (cf. Sect. 1) are easier to adapt to the setting of models of service compositions than approaches in the first category. The MaxSMT approach, the fully flow-sensitive trace formula approach and the constraint satisfaction approach are adaptable to the service setting by enhancing the respective trace formula by additional predicates, which stem from the pre- and postcondition of the single services as well as the overall service composition. Thus, the application of trace formula approaches seems worthwhile to investigate as similar encodings of traces are already in use for verification of service compositions. As service compositions tend to be small, we do not think that the application of error invariants drastically improves the performance of error localization although the method is applicable in general. Dynamic slicing as in [52] gathers information during the execution of programs. As we cannot execute services, but statically compute traces, we suspect dynamic slicing to perform as good as static slicing in our context.

We believe that error localization in service compositions might not only support developers in debugging, but might also be useful to speed up automatic configuration approaches for service compositions. Service compositions tend to be simple. Thus, a systematic approach supporting developers might not be necessary, but when it comes to automatic configuration of service compositions, finding errors will help to only reconfigure erroneous parts and not the overall service composition.

As future work, we plan to examine the proposed modifications to the existing software error localization methods and practically evaluate their effectiveness. Most promising seems to be the use of logical formula-based approaches combined with symbolic executions since the interfaces to services are already given as logical formulas (pre- and postconditions), and the structural aspects of service compositions can easily be encoded by logic.

References

1. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: An evaluation of similarity coefficients for software fault localization. In: 12th Pacific Rim International Symposium on Dependable Computing, PRDC 2006, pp. 39–46 (2006)
2. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, TAICPART-MUTATION 2007, pp. 89–98 (2007)
3. Agarwal, P., Agrawal, A.P.: Fault-localization techniques for software systems: a literature review. SIGSOFT Softw. Eng. Notes **39**(5), 1–8 (2014)

4. Agrawal, H., Demillo, R.A., Spafford, E.H.: Debugging with dynamic slicingand backtracking. Softw. Pract. Exp. **23**, 589–616 (1993)
5. Agrawal, H., Horgan, J.R.: Dynamic program slicing. In: Proceedings of ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, PLDI 1990, pp. 246–256. ACM (1990)
6. Artzi, S., Dolby, J., Jensen, S.H., Moller, A., Tip, F.: A framework for automated testing of javascript web applications. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 571–580 (2011)
7. Artzi, S., Dolby, J., Tip, F., Pistoia, M.: Directed test generation for effective fault localization. In: Proceedings of 19th International Symposium on Software Testing and Analysis, ISSTA 2010, pp. 49–60. ACM (2010)
8. Artzi, S., Dolby, J., Tip, F., Pistoia, M.: Fault localization for dynamic web applications. IEEE Trans. Softw. Eng. **38**(2), 314–335 (2012)
9. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., Ernst, M.D.: Finding bugs in web applications using dynamic test generation and explicit-state model checking. IEEE Trans. Softw. Eng. **36**(4), 474–494 (2010)
10. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: Proceedings of 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2003, pp. 97–105. ACM (2003)
11. Becker, S., Koziolek, H., Reussner, R.: The palladio component model for model-driven performance prediction. J. Syst. Softw. **82**(1), 3–22 (2009). Special Issue: Software Performance - Modeling and Analysis
12. Becker, S., Grunske, L., Mirandola, R., Overhage, S.: Performance prediction of component-based systems – a survey from an engineering perspective. In: Reussner, R., Stafford, J.A., Ren, X.-M. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 169–192. Springer, Heidelberg (2006)
13. Brüseke, F., Wachsmuth, H., Engels, G., Becker, S.: PBlaman: performance blame analysis based on Palladio contracts. Concurr. Comput. Pract. Exp. **26**(12), 1975–2004 (2014)
14. Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.: Pinpoint: problem determination in large, dynamic internet services. In: International Conference on Dependable Systems and Networks, DSN 2002, Proceedings, pp. 595–604 (2002)
15. Christ, J., Ermis, E., Schäf, M., Wies, T.: Flow-sensitive fault localization. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 189–208. Springer, Heidelberg (2013)
16. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
17. Cleve, H., Zeller, A.: Locating causes of program failures. In: Proceedings of 27th International Conference on Software Engineering, ICSE 2005, pp. 342–351. ACM (2005)
18. Crnkovic, I., Chaudron, M.R.V., Larsson, S.: Component-based development process and component lifecycle. In: ICSEA, p. 44. IEEE Computer Society (2006)
19. Dallmeier, V., Lindig, C., Zeller, A.: Lightweight defect localization for Java. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 528–550. Springer, Heidelberg (2005)
20. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Heidelberg (2014)

21. Engels, G., Güldali, B., Soltenborn, C., Wehrheim, H.: Assuring consistency of business process models and web services using visual contracts. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 17–31. Springer, Heidelberg (2008)
22. Ermis, E., Schäf, M., Wies, T.: Error invariants. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 187–201. Springer, Heidelberg (2012)
23. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. Int. J. Softw. Tools Technol. Transf. **8**(3), 229–247 (2006)
24. Groce, A., Kroening, D.: Making the most of BMC counterexamples. Electron. Notes Theor. Comput. Sci. **119**(2), 67–81 (2005)
25. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: 16th International Conference on Software Engineering, Proceedings, ICSE-16, pp. 191–200 (1994)
26. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of 20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005, pp. 273–282. ACM (2005)
27. Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: Proceedings of 24th International Conference on Software Engineering, ICSE 2002, pp. 467–477. ACM (2002)
28. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. SIGPLAN Not. **46**(6), 437–446 (2011)
29. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Proceedings of 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 437–446. ACM (2011)
30. Könighofer, R., Bloem, R.: Automated error localization and correction for imperative programs. In: Proceedings of International Conference on Formal Methods in Computer-Aided Design, FMCAD 2011, pp. 91–100. FMCAD Inc. (2011)
31. Korel, B., Laski, J.: Dynamic program slicing. Inf. Process. Lett. **29**(3), 155–163 (1988)
32. Lamraoui, S.-M., Nakajima, S.: A formula-based approach for automatic fault localization of imperative programs. In: Merz, S., Pang, J. (eds.) ICFEM 2014. LNCS, vol. 8829, pp. 251–266. Springer, Heidelberg (2014)
33. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. SIGPLAN Not. **38**(5), 141–154 (2003)
34. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: Proceedings of ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI 2003, pp. 141–154. ACM (2003)
35. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. In: Proceedings of 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 15–26. ACM (2005)
36. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. SIGPLAN Not. **40**(6), 15–26 (2005)
37. Mao, C.: Slicing web service-based software. In: 2009 IEEE International Conference on Service-Oriented Computing and Applications (SOCA), pp. 1–8 (2009)
38. Minamide, Y.: Static approximation of dynamically generated web pages. In: Proceedings of 14th International Conference on World Wide Web, WWW 2005, pp. 432–441. ACM (2005)
39. Mirandola, R., Potena, P., Riccobene, E., Scandurra, P.: A reliability model for service component architectures. J. Syst. Softw. **89**, 109–127 (2014)

40. OASIS. Web services business process execution language v2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf
41. Renieres, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: 18th IEEE International Conference on Automated Software Engineering, Proceedings, pp. 30–39 (2003)
42. Schäfer, W., Wehrheim, H.: Model-driven development with Mechatronic UML. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 533–554. Springer, Heidelberg (2010)
43. Vessey, I.: Expertise in debugging computer programs: an analysis of the content of verbal protocols. IEEE Trans. Syst. Man Cybern. **16**(5), 621–637 (1986)
44. Walther, S., Wehrheim, H.: Knowledge-based verification of service compositions - an smt approach. In: 2013 18th International Conference Engineering of Complex Computer Systems (ICECCS), pp. 24–32 (2013)
45. Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., Su, Z.: Dynamic test input generation for web applications. In: Proceedings of 2008 International Symposium on Software Testing and Analysis, ISSTA 2008, pp. 249–260. ACM (2008)
46. Weiser, M.: Program slicing. In: Proceedings of 5th International Conference on Software Engineering, ICSE 1981, pp. 439–449. IEEE Press (1981)
47. Wong, W.E., Debroy, V.: A survey of software fault localization. Technical report, The University of Texas at Dallas (2009)
48. Wotawa, F., Nica, M., Moraru, I.: Automated debugging based on a constraint model of the program and a test case. J. Logic Algebraic Program. **81**(4), 390–407 (2012). Special Issue: NWPT 2009Special Issue: NWPT 2009
49. Zeller, A.: Yesterday, my program worked. Today, it does not. Why? In: Wang, J., Lemoine, M. (eds.) ESEC 1999 and ESEC-FSE 1999. LNCS, vol. 1687, pp. 253–267. Springer, Heidelberg (1999)
50. Zeller, A.: Isolating cause-effect chains from computer programs. In: Proceedings of 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT 2002/FSE-10, pp. 1–10. ACM (2002)
51. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. **28**(2), 183–200 (2002)
52. Zhang, X., He, H., Gupta, N., Gupta, R.: Experimental evaluation of using dynamic slices for fault location. In: Proceedings of Sixth International Symposium on Automated Analysis-driven Debugging, AADEBUG 2005, pp. 33–42. ACM (2005)