Fault Localization in Service Compositions*

Heike Wehrheim

Department of Computer Science
Paderborn University, Germany
wehrheim@upb.de

Abstract. Service-oriented architectures aim at the assemblance of service compositions out of independent, small services. In a model-based design, such service compositions first of all exist as formal models, describing the employed services with their interfaces plus their assemblance. Such formal models allow for an early analysis with respect to user requirements. While a large number of such analysis methods exist today, this is less so for techniques localizing *faults* in erroneous service compositions.

In this paper, we extend an existing technique for fault localization in software to the model-based domain. The approach employs *maximum satisfiability solving* (MAX-SAT) of trace formulae encoding faulty program runs. Contrary to software, we can, however, not use testing as a way of determining such faulty runs, but instead employ SMT solving. Moreover, due to the usage of undefined function symbols encoding concepts of domain ontologies, the trace formula also needs an encoding of the logical structure producing the fault. We furthermore show how to use *weights* on soft constraints in the MAX-SAT instance to steer the solver to particular types of faults.

1 Introduction

Service-oriented architectures (SOA) aim at the configuration of service compositions out of existing, independently deployable services. With micro services, this fundamental principle has recently found new impetus. In a formal model-based design, the employed services are equipped with well-defined interface descriptions, and the composition can hence be checked with respect to user requirements before assemblance. Many such analysis methods for service compositions exist today (e.g. [10,20,5,9,21,17]), using a variety of formalisms for modelling.

While formal analysis techniques are numerous, this is less so for techniques localizing the *cause* of faults in erroneous compositions. Analysis techniques building on model checking return counter examples, but they do not give hints on whether to change the way of composition or to change services, and in case

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

of the latter which service to change. For software, this is a more frequently studied question. Techniques like delta debugging [25,7], slicing [24,1] or statistical methods [16] all aim at localizing the cause of faults in programs. Performance blame analysis [6] targets the identification of software components causing long runtimes.

In [15], Krämer and Wehrheim studied the applicability of software error localization techniques to service compositions. It turned out that it is in general difficult to transfer such techniques to a model-based setting: Software fault localization typically requires the executability of the entity under examination, i.e., requires code or requires entities with some accompanied interpretation, compilation or simulation tool as to execute them. Programs are being run on one or – in case of statistical methods – even a large number of test cases, and the outcome of these tests is used to determine potential error locations. Models of service compositions, however, typically cannot be executed as no concrete service implementations might exist yet, and modelling languages often do not come with interpreters providing some form of execution.

A notable exception to the requirement of executability in fault localization is the work of Jose et al. [13]. While they also study programs and make use of testing, they in addition employ bounded model checking for test input generation. This is similar to the SMT-based verification technique which we already employ for service compositions [23]. The basic principle of Jose et al. is the encoding of faulty program runs as trace formulae, and the localization of likely fault causes (in the faulty program trace) via maximum satisfiability solving (MAX-SAT).

In this paper, we transfer this concept to the model-based setting. The main challenge for this is directly the fact that we work on the level of models: instead of (only) using the typical programming language types, service interfaces are formulated via the concepts of *domain ontologies*, and service assemblance uses predicates and rules of these ontologies. Semantically, such concepts and predicates have no fixed interpretation; only ontology rules sometimes slightly restrict their meaning. A faulty test input (as required by Jose et al.) can thus not simply be an input variable of the program with its value. A trace formula encoding a faulty run cannot simply accumulate the concrete statements in the run, but needs to incorporate service calls of which just the interfaces are known. Technically, concepts and predicates of ontologies will in our approach be encoded as undefined function symbols, and a test input is then a complete logical structure fixing an interpretation of these symbols together with a valuation of variables in this logical structure. On this basis, we construct (1) logical formulae encoding the query for correctness of the whole service composition based on a given domain ontology and (2) trace formulae encoding faulty runs in case of errors based on the logical structure used for the fault. Satisfiability of these formulae are solved using the SMT solver Z3 [18]. Out of the trace formula, we can derive *fault sets* via maximum satisfiability solving, i.e., locations in the service composition which – when changed – correct errors. We furthermore allow to *prioritize* fault types, e.g., when we prefer to see faults in service calls over faults

in the structure of the service composition. To this end, we impose weights on constraints in the MAX-SAT instance. Finally, we suggest a technique for finding more likely faults by identifying service composition entities which are the cause of faults in several cases.

The technique we propose here identifies single entities in service compositions as faults. It thus complements the techniques introduced by Krämer and Wehrheim [14] which aim at a repair of faulty service compositions via the replacement of large parts of the composition.

The paper is structured as follows. In the next section, we introduce some necessary background in logic and introduce the model of service compositions we employ in this paper. Section 3 describes correctness checking for service composition which is the basis for identifying faulty service compositions. The following section then introduces our technique for fault localization, for prioritization of fault types and the computation of likely faults. The last section concludes. Our description of the employed techniques is almost always given in terms of Z3 syntax, i.e., we give logical formulae in the SMT-Lib format [4].

2 Background

We start with describing the necessary background in logics, services and service composition. For the latter, we follow the definitions of Walther and Wehrheim [23].

A service composition describes the assemblance of services from a specific domain (e.g., tourism or health domain). The concepts occurring in such a domain and their interrelations are typically fixed in an ontology. Ontologies can be given in standardized languages, like for instance OWL [2]. Here, we simply fix the ingredients of ontologies[1].

Definition 1. *A* rule-enhanced ontology $K = (\mathcal{T}, \mathcal{P}, \mathcal{F}, R)$ *consists of a finite set of* type *(or concept) symbols* $\mathcal{T}$ *together with constants of these types, a finite set of* predicate *symbols* $\mathcal{P}$*, where every* $p \in \mathcal{P}$ *denotes an* n*-ary relation on types, a finite set of* function *symbols* $\mathcal{F}$ *and a set* R *of* rules *which are predicate logic formulae over some variables* Var *and the constants using* $\mathcal{P}$ *and* $\mathcal{F}$*. We use* $\Phi_K(Var)$ *to describe the set of such formulae.*

Here, we implicitly assume the types to contain integers and booleans with the usual constants. Rules, which are not part of every ontology language, are used to describe relationships between concepts or properties of predicates (e.g., commutativity). An example of a rule language for ontologies can for instance be found in [12].

The running example of our paper operates in the tourism domain, containing types like *Restaurant*, *Rating* and *Location* and a constant $R0$ of type *Restaurant*.

[1] For simplicity, subconcept relations of ontologies are not considered here.

On these, the ontology contains two predicates and two functions:

$$\begin{aligned} \mathit{isHigh} &: \mathit{Rating} \to \mathbf{bool} \\ \mathit{isVegan} &: \mathit{Restaurant} \to \mathbf{bool} \\ \mathit{ratOf} &: \mathit{Restaurant} \to \mathit{Rating} \\ \mathit{locOf} &: \mathit{Restaurant} \to \mathit{Location} \end{aligned}$$

For simplicity, the knowledge base contains just two rules specifying properties of the constant $R0$: $\mathit{isHigh}(\mathit{ratOf}(R0))$ and $\mathit{isVegan}(R0)$.

With its types, an ontology defines – in a logical sense – a large number of *undefined function symbols*. It remains undefined what a *Restaurant* really is; it is just a name. The interpretation of these symbols and the universes of the types are not defined. A logical structure (or *logical model*) fixes this interpretation.

Definition 2. *Let $K = (\mathcal{T}, \mathcal{P}, \mathcal{F}, R)$ be an ontology. A* logical structure *over K, $\mathcal{S}_\mathcal{K} = (\mathcal{U}, \mathcal{I})$, consists of*

- $\mathcal{U} = \bigcup_{T \in \mathcal{T}} \mathcal{U}_T$ *the* universe *of values,*
- $\mathcal{I}$ *an* interpretation *of the predicate and function symbols, i.e., for every $p \in \mathcal{P}$ of type $T_1 \times \ldots \times T_n \to \mathbf{bool}$ and every $f \in \mathcal{F}$ of type $T_1 \times \ldots \times T_n \to T$ we have a predicate*

$$\mathcal{I}(p) : \mathcal{U}_{T_1} \times \ldots \times \mathcal{U}_{T_n} \to \mathcal{U}_{\mathbf{bool}}$$

and a function

$$\mathcal{I}(f) : \mathcal{U}_{T_1} \times \ldots \times \mathcal{U}_{T_n} \to \mathcal{U}_T,$$

respectively, and
- *we require the logical structure to satisfy all rules of the ontology.*

Ontologies provide the concepts which can be used to describe the functionality of services. A *service signature* $(\mathit{Svc}, \mathcal{T}_{in}, \mathcal{T}_{out})$ over an ontology K first of all specifies the *name Svc* of the service as well as the type of its input $\mathcal{T}_{in} \in \mathcal{T}$ and the type of its output $\mathcal{T}_{out} \in \mathcal{T}$. We restrict ourselves to services with single inputs and outputs. A service description in addition adds semantical information to this interface.

Definition 3. *A* service description $(\mathit{sig}, I, O, \mathit{pre}_{\mathit{Svc}}, \mathit{post}_{\mathit{Svc}})$ *of a service named Svc over an ontology K consists of a service signature $\mathit{sig} = (\mathit{Svc}, \mathcal{T}_{in}, \mathcal{T}_{out})$, input variable i and output variable o, a precondition $\mathit{pre}_{\mathit{Svc}}$ over the input variable and a postcondition $\mathit{post}_{\mathit{Svc}}$ over input and output variable, both elements of $\Phi_K(\{i, o\})$.*

The interface of a service thus states what the service requires to hold true upon calling it (preconditions) and what it guarantees when it has finished (postcondition). Services can be assembled by means of a workflow language. While there are different languages in use (e.g. WS-BPEL [19]), we simply employ a programming language like notation here.

Name : RestaurantChecker
Inputs : in with $type(in) = Restaurant$
Outputs : out with $type(out) = Restaurant$,
loc with $type(loc) = Location$
Services : $GetRating : Restaurant \to Rating$,
$GetLocation : Restaurant \to Location$
Precondition : true
Postcondition: $isHigh(ratOf(out)) \wedge isVegan(out)$

```
1  if isVegan(in) then
2  |  r := GetRating(in);
3  |  if ¬isHigh(r) then
4  |  |  out := in;
5  |  else
6  |  |  out := R0;
7  |  fi
8  else
9  |  out := R0;
10 fi
11 loc := GetLocation(out);
```

Fig. 1. Service composition RestaurantChecker

Definition 4. *Let $K = (\mathcal{T}, \mathcal{P}, \mathcal{F}, R)$ be an ontology and Var a set of variables typed over $\mathcal{T}$. The syntax of a* workflow W over K *is given by the following rules:*

$$W ::= Skip \mid u := t \mid W_1; W_2 \mid u := Svc(v)$$
$$\mid \textbf{if } B \textbf{ then } W_1 \textbf{ else } W_2 \textbf{ fi} \mid \textbf{while } B \textbf{ do } W \textbf{ od}$$

with variables $u, v \in Var$, expression t of type $type(u)$, $B \in \Phi_K(Var)$, and Svc a service name.

Herein, we allow for standard expressions on integers and booleans. A complete service composition then consists of such a workflow together with a name, its inputs and outputs, a list of the employed services and their pre- and postcondition.

Figure 1 shows our running example of a service composition. In this, we use two services: *GetRating* with precondition $true$ and postcondition $r = ratOf(in)$ and *GetLocation* with precondition $true$ and postcondition $loc = locOf(out)$ (already instantiated to the variables used in the workflow).

A service composition is furthermore equipped with an overall pre- and postcondition. These pre- and postconditions constitute requirements on the service composition: whenever the whole composition is started with an input satisfying the precondition, it should terminate and at the end satisfy the postcondition. Our example service composition is not correct in that sense: whenever the input to the composition is a vegan restaurant (satisfying $isVegan(in)$) with a low rating (satisfying $\neg isHigh(r)$ for r being the rating of in), the service composi-

tion will output the value of *in* which does not satisfy the (first clause of the) postcondition.

For defining this in a more formal way, we furthermore need the definition of a state: given a logical structure $\mathcal{S}$, a *state in* $\mathcal{S}$ is a mapping from variables to values of the universe, $\sigma : Var \to \mathcal{U}$. In [23], a semantics for workflows is given which is parameterised in a logical structure: for a workflow W, $[\![W]\!]_{\mathcal{S}}$ is a mapping from a set of states in $\mathcal{S}$ to sets of states. We will not repeat the definition here, instead we concentrate on verification and fault localization. The semantics of service calls of Walther and Wehrheim [23] fixes (a) a service to be executable only when the precondition holds in the current state, otherwise it blocks, and (b) the after-state to satisfy the postcondition. As the postcondition is just a logical expression, there may be more than one after-state satisfying the postcondition. A service call thus introduces nondeterminism. The property (a) will play no role in the following; for a technique for computing precondition violations on service calls see [14].

Definition 5. *A service composition with workflow W is* correct *with respect to pre- and postconditions pre and post if the following holds for all logical structures $\mathcal{S}$ for K:*

$$[\![W]\!]_{\mathcal{S}}([\![pre]\!]_{\mathcal{S}}) \subseteq [\![post]\!]_{\mathcal{S}},$$

where for a formula $p \in \Phi_K$ without free variables, we let $[\![p]\!]_{\mathcal{S}}$ denote the set of states in $\mathcal{S}$ satisfying p.

This is a standard partial correctness[2] definition: whenever we start the service composition in a state satisfying the precondition, the state reached when the service composition is completed should satisfy the postcondition. The task of fault localization now requires finding the locations in the workflow which are the cause of incorrectness.

3 Correctness checking

In our setting, fault localization directly builds on the results of correctness checking. Therefore, we next present our technique for checking correctness of service compositions. In its basic approach, we follow Walther and Wehrheim [22] here, however, with some adaptation to fit the approach to the later fault localization. The main adaptation concerns *naming*: while Walther and Wehrheim [22] build one formula for the whole service composition, we build separate formulae for the parts of the composition (i.e. one formula for a condition in an if statement, one for a service call and so on). The formulae are assigned different names, and we can later use these names for the hard and soft constraints in the MAX-SAT instance.

[2] Total correctness would involve defining termination functions for loops.

```
1  if isVegan(in) then
2  |  r := GetRating(in);
3  |  if ¬isHigh(r) then
4  |  |  out1 := in;
5  |  else
6  |  |  out2 := R0;
7  |  end
8  |  out3 := φ(out1, out2)
9  else
10 |  out4 := R0;
11 end
12 out := φ(out3, out4);
13 loc := GetLocation(out);
```

Fig. 2. Service composition RestaurantChecker in SSA-form

3.1 Brief introduction to Z3 syntax

We start with a very brief description of the Z3 syntax which we employ.

Declarations We use three types of declarations: (1) Declaring new sorts (types), written as `(declare-sort <name>)`, (2) declaring function symbols, written as `(declare-fun <name> <signature>)`, and (3) declaring constants, written as `(declare-const <name> <sort>)`. The declared sorts, functions and constants can from then on be used. Note that we do not fix the interpretation of these new concepts.

Definitions Definitions, written as `(define-fun <name> <sig> <def>)`, allow to give an interpretation to functions.

Assertions Assertions fix the facts that we would like to hold true. When asked for satisfiability, Z3 checks whether all assertions jointly can be made true. This involves trying to find an interpretation for the unknown (but declared and used) concepts.

Logic Z3 uses standard predicate logic for writing logical expressions (e.g., quantifiers, boolean connectives). The connective `ite` stands for a conditional expression (if-then-else).

3.2 Encoding the service composition

Checking the correctness of a service composition proceeds in two steps: in a first step, we bring the service composition into static-single-assignment (SSA) form. Second, we translate the workflow, its precondition, the ontology and the negation of the postcondition into a logical formula which is then checked for satisfiability. We thereby check whether it is possible to start in a state satisfying the precondition but end in a state not satisfying the postcondition.

SSA forms require a variable to be assigned to at most once in the program text (for a technique for computing SSA forms see [8]). Our service composition

Statement	Z3 code
k: *Skip*	–
k: $u := t$	`(define-fun ak () Bool (= u t))` `(assert ak)`
k: $u := Svc(v)$	`(define-fun ak () Bool post`$_{Svc}$`(u,v))` `(assert ak)`

k: **if** B **then** W_1 **else** W_2 **fi**;
$u := \phi(v_1, v_2)$
	`(declare-const condk Bool)`
	`(define-fun ck () Bool (= condk B))`
	`(define-fun branchk () Bool (= u (ite condk v1 v2)))`
	`(assert ck)`
	`(assert branchk)`

k: inv : **while** B **do** W **od**;
$u := \phi(v_1, v_2)$
	`(declare-const condk Bool)`
	`(define-fun ck () Bool (= condk B))`
	`(define-fun loopk () Bool (= u (ite condk v1 v2)))`
	`(assert ck)`
	`(assert loopk)`
	`(define-fun invk () Bool (and inv(u) (not B(u))))`
	`(assert invk)`

Table 1. Translation of workflow statements

is not yet in SSA form, as it for instance has three assignments to *out*. Figure 2 shows its SSA form. At merges of control flow (if, while) so called ϕ-nodes (assignments) are inserted in order to unite variables assigned to in different branches.

Given such an SSA-form, the translation next proceeds as follows. Table 1 shows the translation of the workflow (directly given in Z3 syntax, the solver we use). Every statement of the workflow is translated to a function declaration and this declaration is asserted to hold true. The thereby introduced names for statements will later prove helpful for fault localization. For assignments, we assert the variable to be equal to the assigned expression. For service calls, we assert the postcondition of the service to be true (as said before, precondition violations are not checked here). In the definition, the term `post`$_{Svc}$`(u,v))` refers to the logical formula as specified for the postcondition in the service description.

For if statements, we get two function declarations: one giving a name to the condition and the other setting variables to the correct version according to the condition and the ϕ-nodes. In the table, we assume every if and while statement to be followed by just one ϕ-node. The translation can, however, easily be generalized to more than one such node. The ϕ-nodes are translated to assertions equating the assigned variable to one of the parameters, based on the condition in the if and while statement, respectively. For loops, we assume a *loop invariant* to be given (see [23]). The invariant together with the negation of the loop condition acts like a postcondition of the whole loop block. The invariant is checked to actually be an invariant in a separate step. The assertion for a loop thus states that the invariant and the negation of the loop condition holds after

Entity	Z3 code
type $T \in \mathcal{T}$	`(declare-sort T)`
constant $c : T$	`(declare-const c T)`
predicate $p : T_1 \times \ldots \times T_n \to \mathbf{bool} \in \mathcal{P}$	`(declare-fun p (T1 ... Tn) Bool)`
function $f : T_1 \times \ldots \times T_n \to T \in \mathcal{F}$	`(declare-fun f (T1 ... Tn) T)`
rule $r \in R$	`(assert r)`

Table 2. Translation of ontology $K = (\mathcal{T}, \mathcal{P}, \mathcal{F}, R)$

the loop, where the variables in both formulae have to be instantiated to the *current variable version* as given by the ϕ-node after the loop.

In addition, we need a translation of the ontology. This requires declaring all types, constants, predicates and functions and asserting the rules to hold true. Table 2 gives this translation.

In summary, this translation gives us a formula φ_{SC} for the workflow, a formula φ_K for the ontology and two formulae *pre* and *post* for the translation of the service composition's pre- and postcondition. The translation for service composition RestaurantChecker is given in the appendix. The solver is now queried about the satisfiability of

$$pre \wedge \varphi_{SC} \wedge \varphi_K \wedge \neg post \tag{1}$$

(plus about the satisfiability of $inv \wedge \varphi_W \wedge \neg inv'$ for all loops with invariant inv, loop body W and SSA form tagging variables after the loop with primes). In case of our example service composition, the answer to (1) is "sat", i.e., it is possible that the composition's postcondition is not fulfilled at the end. Together with this answer, the solver returns a logical structure (which we call *fault structure*), i.e., universes for all sorts (types) and interpretations of the undefined predicate and function symbols, plus a state σ mapping all variables in the service composition to values of the universe. This state can be seen as the final state of the service composition, i.e., the state in which the negation of the postcondition is satisfied. Since the service composition is in SSA-form, the state of a variable is, however, never changed anyway (loops are summarized in the loop invariant). For our running example, Figure 3 shows this information; the appendix contains it in the form returned by Z3. The fault structure constitutes our input to fault localization.

4 Fault localization

Fault localization now proceeds by taking the "faulty" input to the service composition, i.e., the universe, interpretation and the state of the input to the service composition, together with the service composition and ontology, and solving a *partial maximum satisfiability* problem. Essentially, we are determining which

$$
\begin{aligned}
\mathcal{U}_{Restaurant} &= \{Res!val!0, Res!val!1\} \\
\mathcal{U}_{Rating} &= \{Rat!val!0, Rat!val!1\} \\
\mathcal{U}_{Location} &= \{Loc!val!0\} \\
isVegan(Res!val!0) &= true \\
isVegan(Res!val!1) &= true \\
isHigh(Rat!val!0) &= true \\
isHigh(Rat!val!1) &= false \\
ratOf(Res!val!0) &= Rat!val!0 \\
ratOf(Res!val!1) &= Rat!val!1 \\
\sigma(in) &= Res!val!1
\end{aligned}
$$

Fig. 3. "Fault structure" for example service composition

parts of the service composition plus *non-negated* postcondition can simultaneously be satisfied on the faulty input. The complementary part contains the potential faults.

4.1 Computing fault sets

A partial maximum satifiability problem pMAX-SAT takes as input a predicate logic formula in conjunctive normal form. In this, some clauses are marked *hard* (definitely need to be satisfied) and others are marked as *soft* (need not necessarily be satisfied). The pMAX-SAT solver determines a maximum number of clauses which satisfies these constraints. In addition, Z3 – when used as pMAX-SAT solver [11] – allows to give *weights* to soft clauses. A weight sets a penalty for not making a clause satisfied, and Z3 determines solutions with minimal penalties. We will make use of this for prioritizing certain faults over others.

For fault localization, we build a *trace formula* encoding the faulty "run" of the service composition. In our model-based setting, the trace formula needs to contain the whole (logical) fault structure $\mathcal{S}$ and the state σ of the fault. Table 3 shows the translation of this to Z3 input. For the universe, we declare all values of a type and state all variables of that type to just take values of the universe. For predicates and functions, we enumerate all cases of argument and result values. Out of the state, we just fix the value of the input to the service composition. This translation gives us two more formulae: $\varphi_{\mathcal{S}}$ and φ_{σ}. These two formulae describe a structure and state on which the postcondition cannot be satisfied.

Proposition 1. *Let SC be a service composition with pre- and postcondition pre and post, respectively, and K an ontology. Let* $pre \wedge \varphi_{SC} \wedge \varphi_K \wedge \neg post$ *be satisfiable and* $(\mathcal{S}, \sigma)$ *be the logical structure and state making the formula true.*

Entity	Z3 code
universe $\mathcal{U}_T = \{v_1, \ldots, v_n\}$	`(declare-fun v1 T) ... (declare-fun vn T)` `(assert (forall ((x T)) (or (= x v1) ... (= x vn))))`

predicate
$\mathcal{I}(p)(v_1, \ldots, v_n) = true$
$\mathcal{I}(p)(v_1, \ldots, v_n) = false$
	
	`(assert (p v1 ... vn))`
	`(assert (not (p v1 ... vn)))`

function $\mathcal{I}(f)(v_1, \ldots, v_n) = v$	 `(assert (= v (f v1 ... vn)))`
input i, $\sigma(i) = v$	`(assert (= i v))`

Table 3. Translation of logical structure $\mathcal{S}$ and state σ

Then

$$pre \wedge \varphi_{SC} \wedge \varphi_K \wedge post \wedge \varphi_{\mathcal{S}} \wedge \varphi_\sigma$$

is unsatisfiable.

For fault localization, we next ask for partial maximum satisfiability, making all assertions hard except for those of the service composition (the fault is in the service composition). The query thus is (underlining all hard constraints)

$$\underline{pre} \wedge \varphi_{SC} \wedge \underline{\varphi_K} \wedge \underline{post} \wedge \underline{\varphi_{\mathcal{S}}} \wedge \underline{\varphi_\sigma}$$

For every clause in this formula, the solver can tell us whether the clause has or has not been made true. Let $F = \{c_1, \ldots, c_n\}$ be the set of (soft) clauses not made true. F constitutes (one) *fault set*. By changing all assertions (and hence statements) in F, the service composition can be corrected. Similarly to Jose et al. [13], we can also find different fault sets (for the same fault) by adding a new hard constraint $c_1 \vee \ldots \vee c_n$ to our formula.

In case of our example service composition, the solver is always returning a singleton fault set. The fault sets we get by repeatedly starting the solver and adding new hard clauses are $F_1 = \{branch3\}$ (the if statement in line 3 itself), $F_2 = \{cond3\}$ (condition of if statement in line 3), $F_3 = \{cond1\}$ (condition of if statement in line 1), $F_4 = \{a4\}$ (assignment in line 4) and $F_5 = \{a2\}$ (assignment in line 2). A correction of any one of these can make the service composition correct. The easiest way for correction is to change the condition in line 3 to $isHigh(r)$.

4.2 Prioritizing fault types

The MAX-SAT solver typically returns the fault sets in any order, not giving preferences to any soft clause. If we are interested in certain types of faults and prefer to see these first, we need to assign *weights* to soft clauses. Z3 then solves an optimization problem: it tries to find a maximum satisfiability solution which minimizes penalties. A weight therein incurs a penality on not making the clause true (with penalty 1 being standard).

Name : RestaurantChecker2
Inputs : in with $type(in) = Restaurant$
Outputs : out with $type(out) = Restaurant$
Precondition : true
Postcondition: $isVegan(out) \wedge (isCheap(out) \vee isHAcc(out))$

```
1  if ¬(isVegan(in)) then
2  |  if isCheap(in) then
3  |  |  out := in;
4  |  fi
5  |  if isHAcc(in) then
6  |  |  out := in;
7  |  fi
8  else
9  |  out := R0;
10 fi
```

Fig. 4. Service composition RestaurantChecker2

If our interest is thus in localizing faults in conditions or service calls first and delaying more difficult faults in if- or while-statements to later phases, we could tag the corresponding clauses with higher penalties. For our translation we could replace the assertions for if-statements and loops by

```
(assert-soft branchk :weight 2)
(assert-soft invk :weight 3)
```

This gives high penalities to if- and even higher to while-statements. The solver would then first return fault sets with assignments, service calls and conditions. In our case, the fault set F_1 would then be returned as the last set.

4.3 Finding faults occuring multiple times

The previous technique helps to prioritize certain fault types over others. For narrowing down the number of faults to look at, we can also determine the statements occuring as faults in more than one "faulty run", i.e., in our case in more than one logical structure $\mathcal{S}$ making formula (1) true. For this, consider the example service composition in Figure 4.

The service composition RestaurantChecker2 is supposed to check whether the restaurant of input parameter in is vegan plus either cheap ($isCheap$) or accessible to handicapped persons ($isHAcc$). We assume the knowledge base to specify $isCheap(R0)$. Again, the service composition is not correct. Fault localization gives us – now phrased in terms of the service composition, not its translation to SMT code – the two fault sets $F_1^1 = \{\neg(isVegan(in))\}$ and $F_2^1 = \{isHAcc(in)\}$, both based on the same logical structure and state invalidating the postcondition. The state σ with its valuation of the input in describes a "test run" of the service composition passing through lines 1, 2 and 3. We call

$\mathcal{F}^1 = F_1^1 \cup F_2^1$ a *test-specific fault set*. The idea is now to generate different "test inputs" and compute their fault sets.

To this end, we add an assertion forcing the current value of the input to take a different interpretation via predicate complementation. Let $\mathcal{I}$ be the interpretation and σ the state returned by query (1). Let i be the input to the service composition and T its type. We now choose a unary predicate $p : T \to Bool$ from our ontology. If $\mathcal{I}(p)(\sigma(i))$ is true, we add an assertion $\neg(p(i)$, otherwise $p(i)$. For our example, we add

```
(assert (not (isHAcc in)))
```

If formula (1) together with this assertion is still satisfiable, we repeat the fault localization procedure thereby getting new fault sets. If the formula is not satisfiable, we choose a different predicate to be complemented and retry. This way, we get several test-specific fault sets $\mathcal{F}^1, \mathcal{F}^2, \ldots$. The intersection of these test-specific fault sets gives us faults which – when corrected – can correct more than one faulty test input at the same time.

In our case, we get a second test-specific fault set

$$\mathcal{F}^2 = \{\neg(isVegan(in)), isCheap(in)\} \ .$$

Intersection with $\mathcal{F}^1$ gives us $\neg(isVegan(in))$, which is where the correction should be applied.

In general, predicates to be complemented should not be arbitrarily chosen. Predicates occuring in boolean conditions of the service composition are to be preferred over others (as they steer the execution to different branches). If none of the unary predicates bring us new fault sets, arbitrary n-ary predicates can be considered as well.

5 Conclusion

In this paper, we have introduced a fault localization technique for service compositions. It builds on an existing method for *software* fault localization via maximum satisfiability solving. Due to our setting of model-based development, our approach needs to account for several additional aspects: (1) In contrast to software, models are not executable and hence faulty test inputs cannot be derived via testing but only via SMT solving; (2) concepts of ontologies are undefined function symbols in a logical sense and hence have no fixed interpretation; (3) faulty inputs alone are not sufficient for fault localization when no interpretation is known. In addition to plain fault localization, we furthermore provide a method for prioritizing fault types and for finding faults occurring in multiple runs.

This fault localization technique is currently implemented within the modelling and verification tool SeSAME [3]. First experiments show that fault localization can be done within seconds and often just proposes singleton faults which can easily be corrected. However, more experiments are needed to see whether this observation can be generalized to more cases.

References

1. Agrawal, H., Demillo, R., Spafford, E.: Debugging with dynamic slicing and backtracking. Software Practice and Experience 23, 589–616 (1993)
2. Antoniou, G., Harmelen, F.: Web ontology language: OWL. In: Staab, S., Studer, R. (eds.) Handbook on Ontologies, pp. 91–110. International Handbooks on Information Systems, Springer Berlin Heidelberg (2009)
3. Arifulina, S., Walther, S., Becker, M., Platenius, M.C.: SeSAME: modeling and analyzing high-quality service compositions. In: ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014. pp. 839–842 (2014), http://doi.acm.org/10.1145/2642937.2648621
4. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010), http://homepage.cs.uiowa.edu/~tinelli/papers/BarST-SMT-10.pdf
5. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction . Journal of Systems and Software 82(1), 3 – 22 (2009), http://www.sciencedirect.com/science/article/pii/S0164121208001015, special Issue: Software Performance - Modeling and Analysis
6. Brüseke, F., Wachsmuth, H., Engels, G., Becker, S.: PBlaman: performance blame analysis based on Palladio contracts. Concurrency and Computation: Practice and Experience 26(12), 1975–2004 (2014), http://dx.doi.org/10.1002/cpe.3226
7. Cleve, H., Zeller, A.: Locating causes of program failures. In: Proc. of 27th International Conference on Software Engineering. pp. 342–351. ICSE '05, ACM (2005), http://doi.acm.org/10.1145/1062455.1062522
8. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (1991)
9. Engels, G., Güldali, B., Soltenborn, C., Wehrheim, H.: Assuring consistency of business process models and web services using visual contracts. In: AGTIVE. LNCS, vol. 5088, pp. 17–31. Springer (2007), http://dx.doi.org/10.1007/978-3-540-89020-1_2
10. Güdemann, M., Poizat, P., Salaün, G., Ye, L.: VerChor: A Framework for the Design and Verification of Choreographies. IEEE Trans. Services Computing 9(4), 647–660 (2016), http://dx.doi.org/10.1109/TSC.2015.2413401
11. Hoder, K., Bjørner, N., de Moura, L.M.: μZ- an efficient engine for fixed points with constraints. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 457–462. Springer (2011), http://dx.doi.org/10.1007/978-3-642-22110-1_36
12. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: a Semantic Web Rule Language combining OWL and RuleML (2004), http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/
13. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 437–446. ACM (2011), http://doi.acm.org/10.1145/1993498.1993550
14. Krämer, J., Wehrheim, H.: A formal approach to error localization and correction in service compositions. In: Software Technologies: Applications and Foundations - STAF 2016 Collocated Workshops: DataMod, GCM, HOFM, MELO,

SEMS, VeryComp, Vienna, Austria, July 4-8, 2016, Revised Selected Papers. LNCS, vol. 9846, pp. 445–457. Springer (2016), http://dx.doi.org/10.1007/978-3-319-50230-4_35
15. Krämer, J., Wehrheim, H.: A short survey on using software error localization for service compositions. In: Service-Oriented and Cloud Computing - 5th IFIP WG 2.14 European Conference, ESOCC 2016. LNCS, vol. 9846, pp. 248–262. Springer (2016), http://dx.doi.org/10.1007/978-3-319-44482-6_16
16. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: Cytron, R., Gupta, R. (eds.) Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003. pp. 141–154. ACM (2003), http://doi.acm.org/10.1145/781131.781148
17. Mirandola, R., Potena, P., Riccobene, E., Scandurra, P.: A reliability model for service component architectures. Journal of Systems and Software 89(0), 109 – 127 (2014)
18. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer Berlin Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-78800-3_24
19. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Science of Computer Programming 67(2–3), 162–198 (2007)
20. Reynisson, A.H., Sirjani, M., Aceto, L., Cimini, M., Jafari, A., Ingólfsdóttir, A., Sigurdarson, S.H.: Modelling and simulation of asynchronous real-time systems using Timed Rebeca. Sci. Comput. Program. 89, 41–68 (2014), http://dx.doi.org/10.1016/j.scico.2014.01.008
21. Schäfer, W., Wehrheim, H.: Model-Driven Development with Mechatronic UML. In: Graph Transformations and Model-Driven Engineering. LNCS, vol. 5765, pp. 533–554. Springer (2010), http://dx.doi.org/10.1007/978-3-642-17322-6_23
22. Walther, S., Wehrheim, H.: Knowledge-based verification of service compositions – an SMT approach. In: 18th International Conference on Engineering of Complex Computer Systems, ICECCS, 2013. pp. 24–32. IEEE Computer Society (2013)
23. Walther, S., Wehrheim, H.: On-the-fly construction of provably correct service compositions - Templates and proofs. Sci. Comput. Program. 127, 2–23 (2016), http://dx.doi.org/10.1016/j.scico.2016.04.002
24. Weiser, M.: Program slicing. In: Proc. of 5th International Conference on Software Engineering. pp. 439–449. ICSE '81, IEEE Press (1981), http://dl.acm.org/citation.cfm?id=800078.802557
25. Zeller, A.: Yesterday, My Program Worked. Today, It Does Not. Why? In: ESEC/FSE. LNCS, vol. 1687, pp. 253–267. Springer (1999), http://dl.acm.org/citation.cfm?id=318773.318946

A Z3 code

Encoding of the correctness check for RestaurantChecker:

```
; types of the ontology
(declare-sort Res)
(declare-sort Loc)
(declare-sort Rat)
; constants and functions of the ontology
(declare-const R0 Res)
(declare-fun ratOf (Res) Rat)
(declare-fun isHigh (Rat) Bool)
(declare-fun isVegan (Res) Bool)
(declare-fun locOf (Res) Loc)
; variables used in SSA form of service composition
(declare-const r Rat)
(declare-const in Res)
(declare-const out Res)
(declare-const out1 Res)
(declare-const out2 Res)
(declare-const out3 Res)
(declare-const out4 Res)
(declare-const loc Loc)
; variables used for conditions in if's
(declare-const cond1 Bool)
(declare-const cond2 Bool)
; two rules of the knowledge base: restaurant R0 has a high rating and is vegan
(assert (isHigh (ratOf R0)))
(assert (isVegan R0))
; the service composition
(define-fun c1 () Bool (= cond1 (isVegan in)))
(define-fun c2 () Bool (= cond2 (not (isHigh r))))
(assert c1)
(assert c2)
(define-fun a1 () Bool (= r (ratOf in)))
(define-fun a2 () Bool (= out1 in))
(define-fun a3 () Bool (= out2 R0))
(define-fun a4 () Bool) (= out4 R0))
(define-fun a5 () Bool (= loc (locOf out)))
(assert a1) (assert a2) (assert a3) (assert a4) (assert a5)
(define-fun branch2 () Bool (= out3 (ite cond2 out1 out2)))
(define-fun branch1 () Bool (= out (ite cond1 out3 out4)))
(assert branch1) (assert branch2)
; the negated postcondition
(assert (or (not (isHigh (ratOf out))) (not (isVegan out))))
; checking for satisfiability of all asserts
```

```
(check-sat)
; getting an interpretation if it exists
(get-model)
```

Fault structure returned as model of above query:

```
;; universe for Res:
;; Res!val!1 Res!val!0
;; -----------
;; definitions for universe elements:
(declare-fun Res!val!1 () Res)
(declare-fun Res!val!0 () Res)
;; cardinality constraint:
(forall ((x Res)) (or (= x Res!val!1) (= x Res!val!0)))
;; -----------
;; universe for Rat:
;; Rat!val!1 Rat!val!0
;; -----------
;; definitions for universe elements:
(declare-fun Rat!val!1 () Rat)
(declare-fun Rat!val!0 () Rat)
;; cardinality constraint:
(forall ((x Rat)) (or (= x Rat!val!1) (= x Rat!val!0)))
;; -----------
;; universe for Loc:
;; Loc!val!0
;; -----------
;; definitions for universe elements:
(declare-fun Loc!val!0 () Loc)
;; cardinality constraint:
(forall ((x Loc)) (= x Loc!val!0))
;; -----------
(define-fun R0 () Res Res!val!0)
(define-fun out1 () Res Res!val!1)
(define-fun out2 () Res Res!val!0)
(define-fun out () Res Res!val!1)
(define-fun out4 () Res Res!val!0)
(define-fun in () Res Res!val!1)
(define-fun cond2 () Bool true)
(define-fun cond1 () Bool true)
(define-fun out3 () Res Res!val!1)
(define-fun loc () Loc Loc!val!0)
(define-fun r () Rat Rat!val!1)
(define-fun isHigh ((x!0 Rat)) Bool
      (ite (= x!0 Rat!val!0) true (ite (= x!0 Rat!val!1) false true)))
(define-fun ratOf ((x!0 Res)) Rat
```

```
      (ite (= x!0 Res!val!0) Rat!val!0
           (ite (= x!0 Res!val!1) Rat!val!1 Rat!val!0)))
(define-fun isVegan ((x!0 Res)) Bool
     (ite (= x!0 Res!val!0) true (ite (= x!0 Res!val!1) true true)))
(define-fun locOf ((x!0 Res)) Loc
     (ite (= x!0 Res!val!1) Loc!val!0 Loc!val!0)))
```