Subproject B3: Composition Analysis in Unknown Contexts

Matthias Becker[1], Steffen Becker[2], Eyke Hüllermeier[3], Cedric Richter[4], Arnab Sharma[5], Heike Wehrheim[4]

1 Department of Computer Science, Fraunhofer IEM, Paderborn, Germany
2 Department of Computer Science, University of Stuttgart, Stuttgart, Germany
3 Institute of Informatics, LMU Munich, Munich, Germany
4 Department of Computer Science, Oldenburg University, Oldenburg, Germany
5 Department of Computer Science, Paderborn University, Paderborn, Germany

Subproject B3 "Composition Analysis in Uncertain Contexts" deals with the quality assurance of service compositions as assembled by Subproject B2. Over the three funding periods, the objectives varied in the *type* of service composition, *type* of requirements and the *type* of analysis considered.

1 Introduction

Within the CRC, the task of Subproject B3 is to develop techniques for quality assurance of service compositions. Subproject B2 assembles service compositions from services traded on markets based on a requirements specification given by a user. For the interface between Subprojects B2 and B3, a common modeling language has been developed in the first period of the CRC. The task of B3 then consisted of analyzing the service composition before its execution.

In the first period of the CRC, the focus of the subproject has been on

- the development of a common modeling language for describing service compositions as used by Subprojects B1, B2 and B3,
- the analysis of non-functional properties (performance, scalability, and elasticity) of service compositions via simulations, and
- the analysis of functional properties as specified by pre- and postconditions for service compositions via logical encodings and SMT solving.

In the second period, Subproject B3 concentrated on

matthias.becker@iem.fraunhofer.de (Matthias Becker), steffen.becker@informatik.uni-stuttgart.de (Steffen Becker), eyke@lmu.de (Eyke Hüllermeier), cedric.richter@uni-oldenburg.de (Cedric Richter), arnab.sharma@uni-paderborn.de (Arnab Sharma), heike.wehrheim@uni-oldenburg.de (Heike Wehrheim)

- the use of *templates* for service compositions and analysis with the objective of speeding up quality assurance in an *on-the-fly* context,
- the localization of errors in service compositions when requirements are not met (again employing logical encodings and SMT solving), and
- the analysis of non-functional properties via machine learning techniques.

For the third period of the CRC, the focus shifted to considering service compositions with components generated by data-driven techniques. This was motivated by the type of compositions generated by Subproject B2 that started to concentrate on the generation of machine learning (ML) pipelines. Research in B3 then studied

- the analysis of data-driven systems with respect to specific (hyper-)properties,
- machine learning methods for the prediction of non-functional properties of service compositions that can be trained on-the-fly in an online (rather than batch) mode, as well as
- the increase of the robustness of such methods (e.g., against uncertainty or missing information about the context).

Throughout the entire funding period, all conceptual developments were complemented by tool implementations and extensive empirical evaluations. The research results have been published in international conferences and journals. In the following, we highlight some selected results of Subproject B3.

2 Main Contributions

2.1 Performance Prediction via Simulation

In on-the-fly computing scenarios, service compositions were assumed to happen at runtime and on demand. However, those compositions not only need to compose the right services horizontally (i.e., select a complete set of components which together fulfil all domain requirements) but also vertically. The latter means allocating the services on the right amount of resources, i.e., computing, storage and networking capacity.

Nevertheless, in contrast to classical static compositions, the environment of the service composition is unknown and can vary significantly over time. Hence, the allocation needs to adapt to the current environment based on quality requirements expressed via goals and formalized in service level objectives (SLOs).

In our research, we modeled not only these goals, SLOs, but also the composition and the self-adaptations, which always keep adjusting the allocation to the current environment. For these models we have developed a simulator that enables developers to judge the effectiveness of the composition and its self-adaptations upfront. We focused on performance and elasticity in our research and included adaptation strategies in our models and analyses that deal with elasticity.

As introduced above, performance and, more specifically, scalability and elasticity were the quality properties that we aimed to predict based on models of this service composition. For this purpose, a service composition model has to be enriched with performance-relevant annotation. The contributions within Subproject B3 for this are described in the following paragraphs.

Performance Modeling

In order to predict the performance of a service composition, we introduced a performance modeling approach in [BBM13; BLB13] and further refined it with viewpoints and roles in [Bec17]. With our performance modeling approach, we provide the necessary precondition for the assessment of performance properties of service compositions: the extended service composition model contains performance-relevant information, such as the resource consumption of a service, as well as available resources of the service composition's execution environment. Additionally, with service level objectives (SLOs), performance demands for the execution of a service composition can be specified. A concrete workload scenario and its evolution can be specified in order to simulate a service composition execution and thus predict its scalability and elasticity.

Metrics for Scalability and Elasticity

We formally defined a service composition as a self-adaptive system that can be scalable and elastic by adapting its architecture to its performance demands, specified as SLOs, on-the-fly. The formalization is based on the Fuzzy Branching Temporal Logic [MLL04], which allowed us to define a notion of graded SLO achievement, i.e., performance demands of a service can be gradually fulfilled.

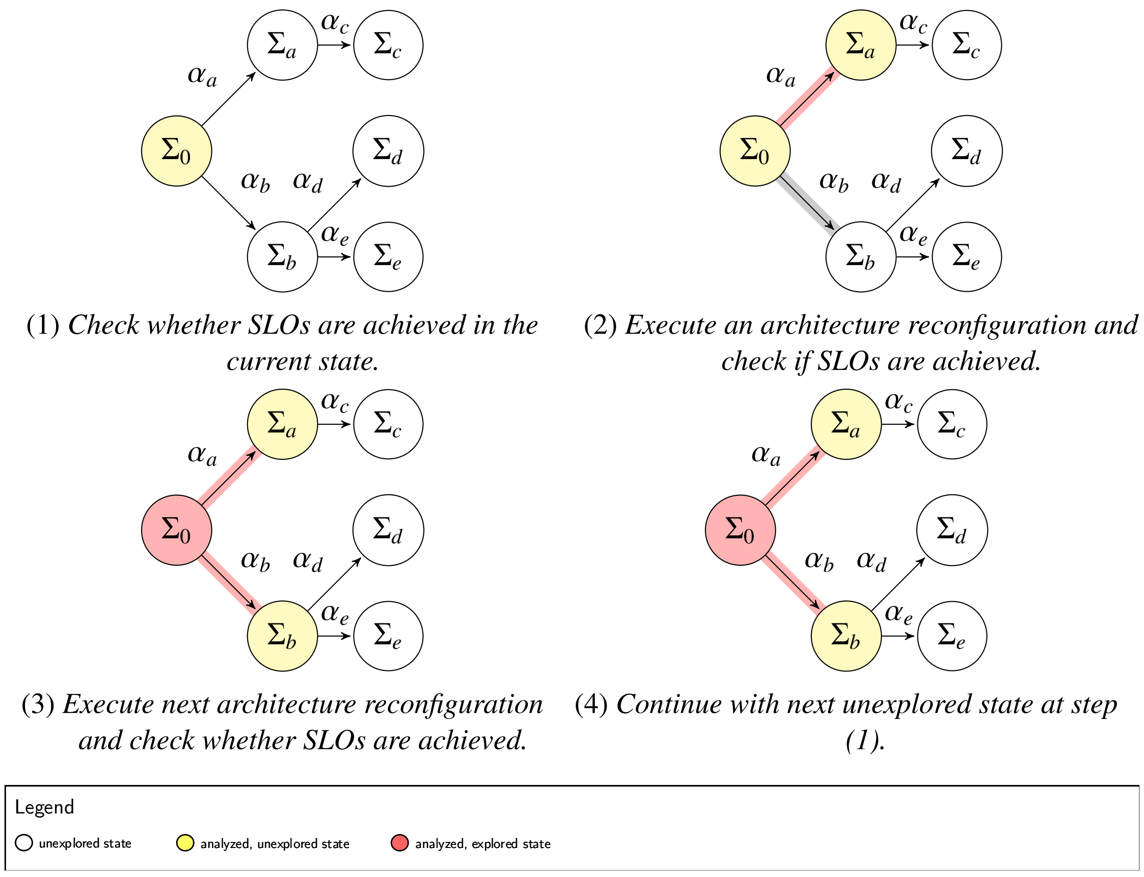
We defined scalability as the ability of a service composition to *eventually* adapt its capacity to different workload scenarios without missing defined service level objectives. Elasticity is the degree to which is service composition is able to self-adapt to workload scenarios, such that it achieves all of its service level objectives to a certain *grade*. To quantify the elasticity grade, we defined the two elasticity metric as *time to SLO achievment (TTSA)* and *accumulated SLO achievement grade (ASAG)*.

TTSA is the metric that reflects the duration a service composition requires to achieve its SLOs in a certain workload scenario. The duration is calculated as the difference from the point in time when the service composition is in a specified state, e.g., its initial state, until the point in time when the service composition is in a state in which its SLOs are achieved. The base unit of the time to SLO achievement is defined as the base unit of time, i.e., seconds (s).

ASAG is the metric that reflects the normalized, accumulated SLO achievement grade of a service composition in a certain workload scenario. The ASAG value is calculated as the (normalized) integral of the SLO achievement of a service composition over time from the point in time when the service composition is in a specified state, e.g., its initial state, until the point in time when the service composition is in a state in which its SLOs are achieved. The metric has no unit, but the values are normalized and are in the interval between 0 and 1, i.e., interval [0; 1].

Prediction of Scalability and Elasticity

Based on our formalization and on our metric definitions, we provided prediction methods for our scalability and elasticity metrics based on a performance simulation of the service composition. Figure 27 illustrates the states of a scalable service composition. The starting state of the simulation Σ_0 is given by the service composition model, see step (1). The state transitions α_i are also defined in the model as model transformations. In order to predict *scalability* of the service composition, each state that is reachable via a model transformation is simulated, see steps (2) and (3) in Figure 27. In each simulation it is checked whether all defined SLOs are achieved eventually, i.e., in a stable performance state. This is repeated until a state is reached that fulfills all SLOs or no more states can be explored, see step (4) in Figure 27. The *elasticity* is predicted by starting a performance simulation in the initial state Σ_0 and applying model transformations during this simulation whenever a precondition of a state transition is met. In this way, the elasticity metrics described above can be determined.



(1) *Check whether SLOs are achieved in the current state.*

(2) *Execute an architecture reconfiguration and check if SLOs are achieved.*

(3) *Execute next architecture reconfiguration and check whether SLOs are achieved.*

(4) *Continue with next unexplored state at step (1).*

Figure 27: *Exploration of scalability.*

2.2 Algorithm Selection for Software Verification

During the second phase of the CRC, Subproject B3 investigated machine learning techniques for *selecting* analysis techniques. More specifically, we looked at various techniques for software verification and studied the question of *algorithm selection*, i.e., how to select an appropriate technique for a verification task at hand [CHJW17; RHJW20; RW19]. Even though software verification is a mature field and a lot of software verification algorithms

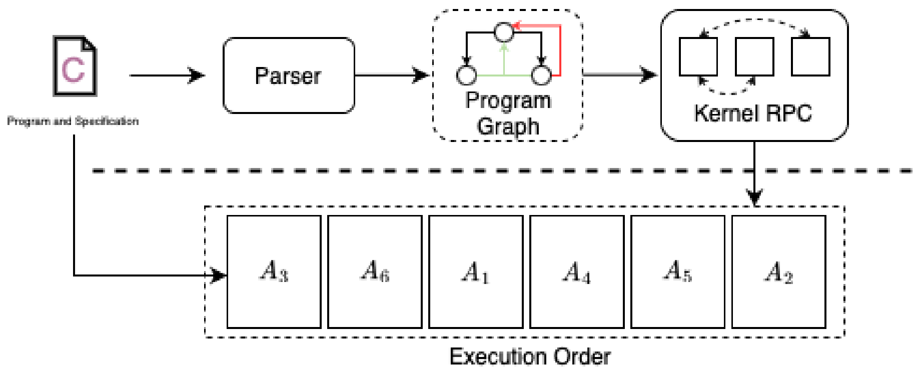


Figure 28: *Overview of the PeSCo framework.*

have been developed over the past decades [BDW15; BF16; BKW10; HCD+13; CJS+16], this is an important question as there is no single algorithm that dominates all other verification algorithms on all possible verification problems. Therefore, we (often manually) have to pick the right algorithm for a given verification task.

To automate the selection process, we developed an approach for predicting the *task-specific* performance of software verification algorithms [RHJW20]. An accurate prediction can then help us to automatically identify and select the best performing algorithm for a given task. In the following, we describe the approach and its instantiations in more detail.

Learning to Select Verifiers

We assume that a verification task consists of a program P and a specification φ. The software verification algorithm, or software verifier for short, then has to verify whether the program satisfies the specification or not. Note that in reality the verification process is limited by system resources and the verifier can only be successful if it verifies a given task within a certain amount of time or memory.

Now, given a set of verifiers $\mathbb{A} = \{A_1, A_2, \dots, A_n\}$, our goal is to identify the verifier that verifies the given verification task within the given resource constraints. For this, we employed a machine learning model that learns to "guess" the performance of the individual algorithms and then rank them accordingly. We then select the highest ranked verification algorithm.

However, to design such a learning based model that can predict the performance of verifiers, we had to overcome two key challenges:

1. How to represent programs and specifications such that we can infer the performance of verifiers?
2. How to integrate our representation into classical machine learning pipelines?

In contrast to previous work [TKK+14; DPVZ17], we decided against representing the verification tasks as feature vectors directly and choose a representation that is closer to the internal representations used inside verifiers. In fact, our approach transforms a given verification task into a combination of an abstract syntax tree, control flow graph and program dependency graph [HR92]. In our case, the specification is encoded inside the program and therefore indirectly represented through the graph structure.

To integrate our program representation into the learning process, we employed a kernel based method [SS02] that enabled us to directly learn on graph representations without an extra feature extraction process. In other words, by employing kernel-based methods, our model learns which graph structures are important for predicting the performance of verifiers. For this, we introduced a custom kernel and utilized kernelized support vector machines [SS02] for the learning process.

During training, our learner learns to rank verification algorithms via the ranking by a pairwise comparison (RPC) framework [FH10]. Here, the learning task is decomposed into multiple binary classification problems. Each resulting classifier then predicts whether a verifier A_i performs better on the given task than another verifier A_j. We define that a verifier A_i is better than a verifier A_j on a given task (and therefore ranked higher) if A_i is more likely to solve the task within the given resource constraints or both verifiers solve the task equally likely but A_i is likely faster.

Finally, we employ the learned model to predict the most likely best performing verifier for a given task. An overview of the prediction process is shown in the upper part of Figure 28. For a new verification task, we first parse the given program and specification into the graph representation. The graph representation is then provided to the learned Kernel RPC model which predicts a ranking of verifiers.

Predicting Sequential Compositions of Verifiers

We implemented our selection approach inside the verification tool CPAchecker [BK11], which ultimately resulted in a new verification tool called PeSCo [RW19]. PeSCo ranks up to six base verification algorithms and then executes them in order. As a result, PeSCo is able to select from over 15 different sequential verifier compositions based on the characteristics of the given verification task.

In addition, we found that performance modeling for ranking verification algorithms is also effective in practice. With its selection approach, PeSCo won the second place in the overall category of the 8th international software verification competition (SVComp) [Bey19] and since then remains highly successful in the competition.

As part of a DFG-funded project on “Cooperative Verification” we continue the work of algorithm selection for software verification, now with a focus on selecting components for a cooperative approach.

2.3 Functional Analysis of Service Compositions

In the first two phases of Subproject B3, we considered the analysis of service compositions specified in the common modeling language, jointly developed between Subprojects B1, B2 and B3 [AWBP14]. The focus was on the analysis of functional properties specified via pre- and postconditions for service compositions. The compositions are assembled out of single services traded on the market. Each such service has a specification written

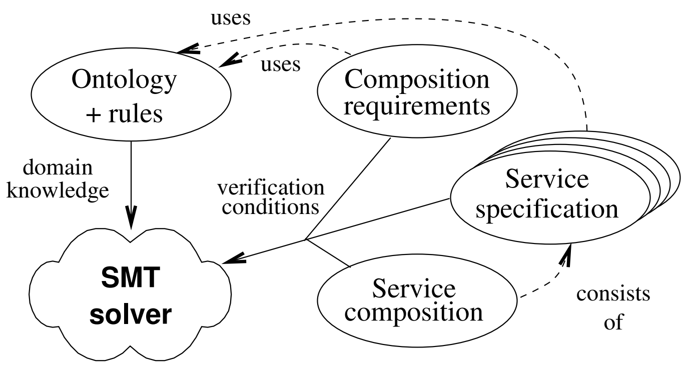


Figure 29: *Overview of the ontology-based approach*

in the modeling language as well. The vocabulary of the modeling language is based on a domain-specific ontology. This accounts to types used in service signatures, but also to predicates occurring in preconditions and effects of services. Ontologies, in particular those enhanced with *rules*, capture the knowledge of domain experts on properties of and relations between domain concepts.

Our verification technique for service compositions [WW13] makes use of this domain knowledge. We consider a service composition to be an assembly of services of which we just know signatures, preconditions, and effects. Compositions are written in a simple workflow language, such as specifiable via activity diagrams. We aim at proving that a composition satisfies a (user-defined) requirement, specified in terms of guaranteed preconditions and required postconditions. For an underlying verification engine we use an SMT solver. More specifically, we translate single service specifications, the service composition and the ontology rules to first order predicate logic to be fed into an SMT solver (see Figure 29). Similarly, we translate the user requirement into a logical formula. To take advantage of the domain knowledge (and often, to *enable* verification at all), the knowledge is fed into the solver in the form of sorts, uninterpreted functions and, in particular, assertions as to enhance the solver's reasoning capabilities. Thereby, we allow for deductions within a domain previously unknown to the solver. In the CRC, we have applied our technique on a case study from the area of water network optimization software (as studied by Subproject C3 on "Modeling of Optimization Problems" in the first phase). In the following, we describe the technique in more detail.

Verification Approach

We assume a given composition of services, each with an ontology-based interface specification. Apart from interfaces, nothing is known about the services (black-box view). In the context of service-oriented architectures (SOA), this is a quite likely scenario: Providers sell their services but not the code itself. In fact, a service might not even run on the consumer side, but could either completely stay on the provider machine or run in the cloud. Furthermore, the requirements on an assembled service composition are specific to the domain; instead of proving general safety or reachability properties alone (as state-of-the-art software verification tools do), consumers expect the verification to prove domain-specific requirements. We leverage this by grounding service specifications on ontologies.

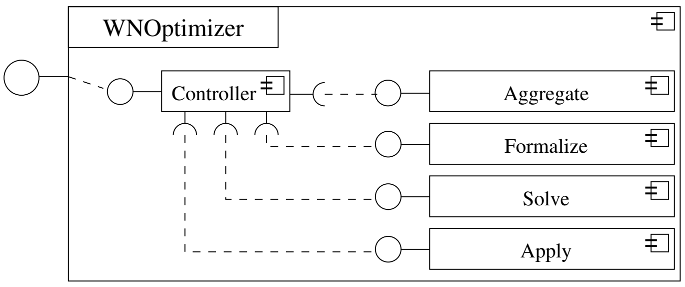


Figure 30: *Service composition of the WaterNet Optimizer.*

We exemplify this with the case study "water supply network optimization" (see Figure 30 for a service composition in this domain). Software services in this domain handle different tasks of analyzing and optimizing existing municipal water supply networks. Single services designed for different subtasks can be assembled into a composition. This concerns services such as (a) compacting the size (and layout) of network models, (b) generating mathematical optimization problems from networks, (c) solving optimization problems, and (d) applying optimal solutions to networks. As the behavior of these services is specified in terms of interfaces only, this is a black-box view and for the analysis we can therefore only assume that services adhere to their specification.

Every water network has specific hydraulic characteristics, as well as other properties such as the cost of operation. A typical domain specific requirement on a composition of some optimization services is that an optimized network (produced by the composition) has the *same* hydraulic characteristics as the original input network, but a *better* (e.g., lower) cost of operation. The verification technique has to derive this property from the given service specifications, the way of assembling the services and the additional domain knowledge stored in the ontology.

Our approach to the verification of such a service composition is based on the use of an SMT solver (satisfiability modulo theories solver) as reasoning engine. Basically, our technique feeds three types of inputs (domain knowledge, service interface specifications, and assembly) into the SMT solver in different forms (see Figure 29). These inputs, combined with the user's requirements specification, are encoded as first-order logic formulae. In this encoding, the user requirement is negated. The resulting formula is then checked for satisfiability: If unsatisfiable, the requirement is fulfilled; if satisfiable, a counterexample is found.

More specifically, we start with an ontology of the domain which – besides the standard concepts and their relations – models additional rules about the domain by first-order logic. The predicates therein are the relations in the ontology. Providers of services use the ontology to specify a service's signature and its preconditions and effects. Consumers use the ontology to specify requirements of a service composition. For verification, we use the concepts of the ontology as *types* for the solver, relations as *uninterpreted functions*, and rules as constraints on the interpretation of these functions. The rules are thus being used for deduction together with the decidable theories of the solver (e.g., linear arithmetic). The creation of verification conditions for a given service composition and requirements follows ideas of Hoare-style proofs. It turns out that the verification typically requires the additional domain knowledge for a successful reasoning: The knowledge of human domain experts (e.g. about hydraulic properties of different forms of networks) needs to be provided to the solver.

Templates

In cooperation with Subproject B2 [MW15], the basic verification approach was complemented with the idea of *templates* [WW14; WW16]. This was motivated by the on-the-fly principle, because pre-verified templates upon instantiation only require checks of the soundness of the instantiation, not of the entire composition. Templates can capture known composition patterns, and thus allow for the application of the general principle of pattern usage in software engineering.

More specifically, following our approach for the verification of service compositions, templates are workflow descriptions with *service placeholders*. Service placeholders are replaced by concrete services during instantiation. If a template is shown to be correct, then all of its (valid) instantiations will be *correct by construction*. Every template specification contains functional properties given in terms of pre- and postconditions (again with associated meaning “if precondition fulfilled, then postcondition guaranteed”), and a correct template provably adheres to this specification. To verify correctness of templates, we employ the Hoare-style proof calculus as of above.

The definition of “correctness” as well as giving a proof calculus for templates, however, poses a non-trivial task on verification. Since templates should be usable in a wide range of contexts and the instantiations of service placeholders are unknown at template design time, we cannot give a fixed semantics to templates. Rather, the template semantics needs to be *parameterized* in usage context and service instantiation. A template is only correct if it is correct for all (allowed) usage contexts. Similarly, a useful proof calculus has to be applicable in all possible contexts and service instantiations. We guarantee this by defining a proof calculus that is *parameterized* in usage contexts and template-specific constraints.

Technically, we capture the usage contexts by ontologies, and the interpretation of concepts and predicates occurring therein by logical structures. A *template ontology* defines the concepts and predicates of a template. Furthermore, a template specification contains constraints defining additional conditions on instantiations. These constraints allow us to verify the correctness of the template despite unknown usage and unknown fixed semantics. A template instantiation replaces the template ontology with a homomorphous domain ontology, and the service placeholders with concrete services of this domain. Verification of the instantiation then amounts to checking whether the (instantiated) template constraints are valid within the domain ontology, and thus can be carried out on-the-fly.

2.4 Performance Prediction via Machine Learning

As an alternative to the use of simulation techniques (cf. Section 2.1), the potential of machine learning (ML) methods for non-functional analysis and performance prediction has been investigated in the second and third funding period. The idea here is to induce models that predict a property of a service composition, given the specification of the service as input. What makes this problem challenging from an ML perspective is the specific structure of service compositions: Services are recursively structured objects of variable size. Representing them in terms of feature vectors of fixed length, the format commonly assumed by most ML methods, is difficult and will necessarily cause a loss of information.

To cope with these challenges, we introduced a new ML setting that we call “learning to aggregate” (LTA). Roughly, learning-to-aggregate problems are supervised machine learning problems in which data objects are represented in the form of a *composition* of a (variable) number on *constituents*; such compositions are associated with an evaluation, score, or label, which is the target of the prediction task, and which can presumably be modeled in the form of a suitable aggregation of the properties of its constituents. Thus, our LTA framework establishes a close connection between machine learning and a branch of mathematics devoted to the systematic study of aggregation functions [GMMP09].

A bit more formally, we proceed from a set of training data $\mathcal{D} = \{(\boldsymbol{c}_1, y_1), \ldots, (\boldsymbol{c}_N, y_N)\} \subset C \times \mathcal{Y}$, where C is the space of compositions and $\mathcal{Y}$ a set of possible (output) values associated with a composition. Since aggregation is often used for the purpose of evaluating a composition, we also refer to the values y_i as *scores*. A composition $\boldsymbol{c}_i \in C$ is a multiset (*bag*) of constituents $\boldsymbol{c}_i = \{c_{i,1}, \ldots, c_{i,n_i}\}$, where $n_i = |\boldsymbol{c}_i|$ is the size of the composition; scores y_i are typically scalar values (e.g., representing a specific non-functional property of a service). Constituents $c_{i,j}$ can be of different type, and the description of a constituent may or may not contain the following information:

- A *label* specifying the role of the constituent in the composition. For example, suppose a composition is a service in the form a machine learning pipeline (cf. Subproject B2) consisting of an algorithm for data preprocessing, a method for inducing a classifier, and an algorithm for postprocessing predictions. By assigning labels to these constituents, such as `pre`, `induce`, and `post`, additional information is provided about the part of the composition they belongs to (thereby adding additional structure to the composition).

- A description of *properties* of the constituent, for example, memory requirements of an algorithm. Formally, we assume properties to be given in the form of a feature vector $\boldsymbol{v}_{i,j} \in \mathcal{V}$, where $\mathcal{V}$ is a corresponding feature space. However, more complex descriptions are conceivable. For example, the description could itself be a composition.

- A *local evaluation* in the form of a score $y_{i,j} \in \mathbb{R}_+$.

Finally, a composition can also be equipped with an additional structure in the form of a (binary) relation on its constituents. In this case, a composition is not simply an unordered set (or bag) of constituents but a more structured object, such as a sequence (like in the above example of an ML pipeline) or a graph.

Like in standard supervised learning, the goal in learning-to-aggregate is to induce a model $h : C \longrightarrow \mathcal{Y}$ that predicts scores for compositions. More specifically, given a hypothesis space $\mathcal{H}$ and a loss function $L : \mathcal{Y}^2 \longrightarrow \mathbb{R}_+$, the goal is to find a hypothesis $h^* \in \mathcal{H}$ that provides optimal predictions in the sense of minimal L in expectation.

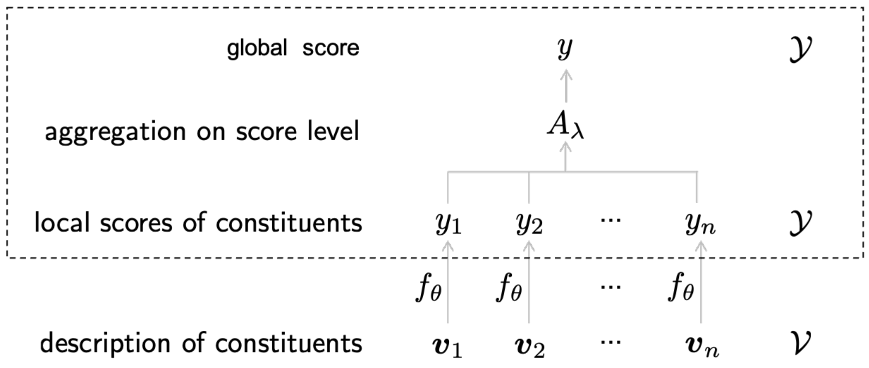

global score
aggregation on score level
local scores of constituents
description of constituents
y
$\mathcal{Y}$
A_λ
y_1 y_2 ⋯ y_n
$\mathcal{Y}$
f_θ f_θ ⋯ f_θ
$\boldsymbol{v}_1$ $\boldsymbol{v}_2$ ⋯ $\boldsymbol{v}_n$
$\mathcal{V}$

Figure 31: *Illustration of a basic version of the learning-to-aggregate framework.*

Learning Aggregation Functions

One of the key problems in learning to aggregate is to combine a variable number of scores $y_{i,j}$, pertaining to evaluations of the constituents $c_{i,j}$ in a composition $\boldsymbol{c}$, into a single score y_i. In Figure 31, which provides an overview of the LTA setting, this step corresponds to the part marked by the dashed rectangle.

Now, suppose that we know or can at least reasonably assume that y_i is obtained from $y_{i,1}, \ldots, y_{i,n_i}$ through an aggregation process defined by a binary aggregation function $A : \mathcal{Y}^2 \longrightarrow \mathcal{Y}$:

$$y_i = A\Big(\ldots A(A(y_{i,1}, y_{i,2}), y_{i,3}), \ldots, y_{i,n_i}\Big).$$

In the simplest case, where the constituents do not have labels and hence cannot be distinguished, the aggregation should be invariant against permutation of the constituents in the bag. Thus, it is reasonable to assume A to be associative and symmetric. Besides, one may of course restrict an underlying class of candidate functions $\mathcal{A}$ by additional assumptions, such as monotonicity.

Starting from a class $\mathcal{A}$ of aggregation functions, instead from a hypothesis space $\mathcal{H}$ directly, has at least two important advantages. First, as just said, it allows for incorporating prior knowledge about the aggregation, which may serve as a suitable inductive bias of the learning process. Second, it naturally solves the problem that hypotheses $h \in \mathcal{H}$ must accept inputs of any size. Indeed, under the assumption of associativity and symmetry, a binary aggregation function A is naturally extended to any arity, and can hence be used as a "generator" of a hypothesis $h = h_A$:

$$h(y_1, \ldots, y_n) = A^{(n)}(y_1, \ldots, y_n) = A\Big(A^{(n-1)}(y_1, \ldots, y_{n-1}), y_n\Big)$$

for all $n \geq 1$, where $h(y_1) = A^{(1)}(y_1) = y_1$ by definition. For these reasons, we consider the learning of (binary) aggregation functions, and related to this the specification of a suitable class $\mathcal{A}$ of candidates, as an integral part of learning to aggregate.

Disaggregation

The aggregation we have been speaking about so far is an aggregation on the level of scores. Thus, we actually assume that local scores $y_{i,j}$ of the constituents $c_{i,j}$ are already given and that we are interested in aggregating them into an overall score y_i of the composition $\boldsymbol{c}_i$. This is indeed the genuine purpose of aggregation functions, which typically assume that all scores are elements of the same scale $\mathcal{Y}$. Now, suppose that local scores $y_{i,j}$ are not part of the training data. Instead, the constituents $c_{i,j}$ are only described in terms of properties in the form of feature vectors $\boldsymbol{v}_{i,j} \in \mathcal{V}$. A natural way to tackle the learning problem, then, is to consider the local scores as latent variables, and to induce them as functions $f : \mathcal{V} \longrightarrow \mathcal{Y}$ of the properties.

More specifically, we assume these functions to be parameterized by a parameter vector θ, and the aggregation function A by a parameter λ. The model is then of the form

$$y_i = A_\lambda(y_{i,1}, \ldots, y_{i,n_i}) = A_\lambda\Big(f_\theta(\boldsymbol{v}_{i,1}), \ldots, f_\theta(\boldsymbol{v}_{i,n_i})\Big) \enspace ,$$

and the problem consists of learning both the aggregation function A, i.e., the parameter λ, and the mapping from features to local scores, i.e., the parameter θ, simultaneously. Here, supervision only takes place on the level of the entire composition, namely in the form of scores y_i, whereas the “explanation” of these scores via induction of local scores is part of the learning problem.

The decomposition of global scores into several local scores is sometimes referred to as *disaggregation* (because it inverts the direction of aggregation, which is from local scores to global ones). One could then try to learn how the constituents are rated (via f_θ) and, simultaneously, how the corresponding local scores are aggregated into a global rating (via A_λ). Obviously, there is a strong interaction between local rating and aggregation on a global level. An important question, therefore, concerns the *identifiability* of the model, i.e., the question whether different parameterizations imply different models (or, more formally, whether $(\lambda, \theta) \neq (\lambda', \theta')$ implies that the corresponding models assign different scores $y_i \neq y_i'$ for at least one composition).

Instantiations

The LTA framework as outlined above has been instantiated in different ways and evaluated on practical learning tasks. A first instantiation based on a class of aggregation functions called *uninorms* has been proposed in [MH16]. Learning algorithms for another type of aggregation function, so-called *ordered weighted averaging operators*, have been developed and tested in [MH19].

2.5 Testing of Data-Driven Software Systems

In the third phase of the CRC, the service compositions to be analyzed by Subproject B3 were *pipelines* of machine learning components, such as data generation, preprocessing, learning etc. generated by Subproject B2. Essentially, through the pipeline of such services, B2 generates data-driven software systems (DSS).

Unlike traditional software systems, where intended behavior of the software is programmed by the developer, data-driven software *learns* its intended functionality from lots of examples. The analysis of such a system faces two fundamental challenges: (1) identification of the requirements to be checked and (2) development of an analysis method.

The first challenge arises out of the fact that the actual intended behavior of the learned component is unclear as otherwise learning would not be required at all. The second challenge arises because learning algorithms generate a diverse set of different classifiers (or regressors).

More precisely, given a set of data instances (also called a *training data set*), a machine learning algorithm generalizes from the data set thereby generating a machine learning (ML) *model* (or DSS[11]). Formally, this model is a mapping from inputs to an output, i.e.,

$$M : X_1 \times \ldots \times X_n \rightarrow Y \, .$$

The X_i denotes the value set of the input element (also called the *feature*) i and Y denotes the set of output values. However, it is essentially unclear what the correct outcome of this process is, i.e., what is considered to be an expected model M. Moreover, even if we can identify some requirements to check, there can be different types of ML models as the outcome of the learning process, depending on the learning algorithm used, such as decision tree, neural network, random forest, support vector machine or others.

In recent years, with the increased usage of such data-driven software, there have been a number of works focusing on ensuring the quality of such data-driven systems (see e.g., [ZHML22; Alb21]). To this end, two approaches are currently followed: a) developing an ML algorithm guaranteeing a requirement per design or b) validating the requirement on a given DSS. There are shortcomings for both of these approaches. Firstly, the requirement-per-design algorithms are only available for a small number of requirements. Moreover, it has been found out that in some cases these algorithms were unable to guarantee the desired requirements [GBM17]). Secondly, validation techniques are either restricted to a specific model or to a specific requirement to check, such as checking fairness for deep neural network model [ZWS+20].

Within Subproject B3, we have proposed a validation technique called *property-driven testing* with the intention of overcoming the shortcomings of existing techniques. Our method is a *validation* technique in that we aim at the falsification of requirements, i.e., finding counterexamples to properties like standard testing techniques do. Contrary to standard testing often using random generation of test inputs, we however have a systematic, verification-based technique for generating potential counterexamples. Our technique is "property-based” as it allows the checking of user-supplied properties, written in a pre- and postcondition format. We have implemented this testing approach in a tool named MLCHECK and have evaluated it to check a number of properties on several types of ML models. All the code and data of this work is publicly available at https://github.com/arnabsharma91/MlCheck. Next, we briefly describe the steps involved in our property-driven testing framework.

Property-Driven Testing

We have developed a testing mechanism that allows the user to specify the property using a standard specification language that would then be used for test case generation. To this end, we have the following two contributions: a) a domain-specific property specification language and b) a targeted test case generation method.

[11]We use the term ML model and data-driven software (DSS) interchangeably in this section.

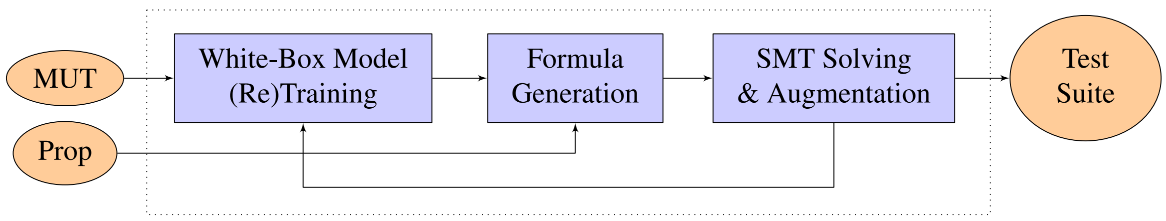


Figure 32: *Workflow of test data generation*

Property specification. We give an *assume/assert* style specification language where an assume statement specifies a condition on the input and assert statement specifies the condition to be satisfied by the output of an ML model. We develop this considering Python as a base programming language because of its high use in developing data-driven software. Essentially, assume/assert statements are defined as calls to the functions `Assume` and `Assert` respectively and take the following form [SDNW21].

```
Assume('<condition>',<arg1>, ...)
Assert('<condition>',<arg1>, ...)
```

The first parameter is a string defining a logical condition on the input data instance (for `Assume`) or the output (for `Assert`) of the model under test (MUT), combining any other variables in the code. The rest of the arguments give the values of these variables respecting the order of their occurrences in the condition. Later, this condition, along with the values of the variables, is translated to a logical formulae. Further details about our specification language and its grammar can be found here [SMHW22].

Test case generation. We perform this step employing a technique called *verification-based* testing, which we propose in [SW20]. To this end, first of all, we generate a set of data instances randomly and for each of these instances we get the corresponding predictions (i.e., outputs) from the MUT. The set of instances, along with their predictions form the training data set for a *white-box model* (see Figure 32). The newly generated white-box model in our framework can be either a decision tree or a neural network model. It approximates the MUT. Next, we convert white-box model W and the negation of the property specification to a logical formula (in SMT-LIB format[12]) ϕ_W and $\phi_{\neg P}$ respectively and we conjunct them to get $\phi_W \wedge \phi_{\neg P}$. This translation to logical formula guarantees to give a satisfiable formula if and only if the white-box model does not satisfy the property. The conjuncted formula then is given to the satisfiability modulo theory (SMT) solver Z3 [MB08]. If the Z3 finds the formula to be satisfiable, it will return a counterexample to the property, i.e., an input to W that shows the violation of the given property.

Now, this counterexample serves as a test case and, using a method called *pruning*, we generate more of these. However, as we find the counterexamples on W, not on the MUT, we must check the validity of the counterexamples on the MUT. In case they are not valid, we add the input instances from the counterexamples, along with its real predictions from MUT to the training data set and retrain the model W to get a better approximation of the MUT. Otherwise, we store the counterexamples and return them as counterexamples for the MUT M. These steps are repeated until a user defined timeout occurs.

[12]http://smtlib.cs.uiowa.edu/

Results

We have implemented the property-driven testing approach in a tool called MLCHECK and applied it to check for several types of properties. For example, in [SW20], we applied our approach to test monotonicity requirements of ML models. Our evaluation shows that our approach outperforms adaptive random testing [CLM04] and property-based testing [CH00] approaches in finding out monotonicity violations. Furthermore, our approach can find out the violations even for ML models that are by designed to be monotonic. We also checked for several types of fairness criteria in [SW20] and found our approach to be effective in finding out more number of fairness violations than the existing fairness testing approaches [ALN$^+$19; UAC18]. Our approach shows that existing learning algorithms that are by design meant to be fair can generate unfair models, leading to fairness violation. In a later work, we furthermore checked security and concept relationship requirements (developed in cooperation with Subproject B2) of data-driven software [SDNW21]. Finally, in a recent work we used MLCHECK to evaluate a number of mathematical properties on a specific type of ML models (i.e., *regression* models) [SMHW22]. In this case, the requirements reflect properties of aggregation functions as studied within B3 in the context of "learning to aggregate" [MH16; MH19]. Thus, we can apply our tool in testing diverse properties for several types of data-driven software systems.

3 Impact and Outlook

The research conducted in this Subproject over the last decade, and notably the contributions highlighted in this chapter, has been impactful and has triggered follow-up work by ourselves and other scholars.

For example, based on our research on the prediction of scalability and elasticity (which ended after the first period due to the leave of PI Becker), several follow-up projects pushed these ideas further. The EU FP7 project CloudScale extended the presented simulation approach to analyze SLO achievement by architectural templates (ATs), which makes it much easier for end users to model typical elasticity patterns in cloud computing allocations. Becker and his colleagues also contributed a pattern catalogue containing patterns for horizontal and vertical scale-up/-down and scale-out/-in including the corresponding load balancing strategies.

When using the approach in practice, it was realized that it can be rather difficult to analyze the simulator's results and to improve the self-adaptation rules based on these results alone. Hence, in a current ongoing DFG project, we aim at explainability of the simulator's results. The vision is that, based on the simulator's results, the system should explain which self-adaptations have been taken when and why. Ideally, it might even make suggestions on how to change the self-adaptation rules to achieve improved results.

Another example of impactful research is our work on algorithm selection for software verification. In particular, the development of the tool PeSCo has inspired the development of other algorithm selectors. We ourselves have shown that approaches based on neural networks can be used to learn *transferable* feature representation, applicable to many verifier selection scenarios [RW20]. Apart from us, Beyer et al. [BKR22] have found that *combinations* of complete verification tools chosen via algorithm selection significantly outperform the performance of single tools. Finally, a new verification tool called GraVeS [LD22] has been developed based on the PeSCo architecture, and has already been evaluated successfully in the software verification competition [Bey22].

Our work on machine learning for predictive modeling of service properties has triggered follow-up work, too. In particular, the "learning-to-aggregate" setting that we introduced has inspired other researchers. Obviously, this setting is not restricted to the prediction of properties of service compositions, but can also be applied to other learning tasks, where global scores are naturally modeled as an aggregation of local evaluations. In [PTF+21], for example, the LTA framework has been picked up and extended by the introduction of so-called learnable aggregation functions (LAF) for sets of any cardinality. This class of functions is shown to be very versatile and able to approximate many important aggregators in a flexible way. In experimental studies, the approach has been compared to other methods for learning from sets, and was found to outperform state-of-the-art approaches from the field of deep neural networks.

On a broader scale, the importance of the research topics addressed in this subproject is even likely to increase in the near future, especially due to the rapid development in the field of artificial intelligence. With the quick expansion of practical AI applications, along with the increasing trend toward the data-driven construction of AI tools based on neural network technology, the verification of these tools is becoming more and more crucial. We initialized work in this direction in the third funding period, but of course, this can mark just the start of a bigger research program. Currently, for example, there is a lot of work on formal verification of neural networks, motivated by the need to provide formal guarantees on the correctness, safety, robustness, or fairness of such networks. In a sense, verification goes hand in hand with other approaches aimed at increasing the trustworthiness of AI systems, such as explainability. We believe that the methods and tools developed in this subproject provide a suitable basis for further developments in this field.

Bibliography

[Alb21] Albarghouthi, A.: Introduction to Neural Network Verification. In: *Found. Trends Program. Lang.* 7 (2021), no. 1-2, pp. 1–157.

[ALN+19] Aggarwal, A.; Lohia, P.; Nagar, S.; Dey, K.; Saha, D.: Black box fairness testing of machine learning models. In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*. Ed. by Dumas, M.; Pfahl, D.; Apel, S.; Russo, A. ACM, 2019, pp. 625–635.

[AWBP14] Arifulina, S.; Walther, S.; Becker, M.; Platenius, M. C.: SeSAME: modeling and analyzing high-quality service compositions. In: *ASE*. Ed. by Crnkovic, I.; Chechik, M.; Grünbacher, P. ACM, 2014, pp. 839–842.

[BBM13] Becker, M.; Becker, S.; Meyer, J.: SimuLizar: Design-Time modeling and Performance Analysis of Self-Adaptive Systems. In: *Proceedings of the Software Engineering Conference (SE)*. Lecture Notes in Informatics (LNI). 2013, pp. 71–84

[BDW15] Beyer, D.; Dangl, M.; Wendler, P.: Boosting k-Induction with Continuously-Refined Invariants. In: *Computer Aided Verification - 27th International Conference, CAV*. Ed. by Kroening, D.; Pasareanu, C. S. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 622–640.

[Bec17] Becker, M.: Engineering Self-Adaptive Systems with Simulation-Based Performance Prediction. PhD thesis. Universität Paderborn, Softwaretechnik, 2017

[Bey19] Beyer, D.: Automatic Verification of C and Java Programs: SV-COMP 2019. In: *Tools and Algorithms for the Construction and Analysis of Systems TACAS TOOLympics, Held as Part of ETAPS Part III*. Ed. by Beyer, D.; Huisman, M.; Kordon, F.; Steffen, B. Vol. 11429. Lecture Notes in Computer Science. Springer, 2019, pp. 133–155.

[Bey22] Beyer, D.: Progress on Software Verification: SV-COMP 2022. In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*. Ed. by Fisman, D.; Rosu, G. Vol. 13244. Lecture Notes in Computer Science. Springer, 2022, pp. 375–402.

[BF16] Beyer, D.; Friedberger, K.: A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker. In: *Proceedings 11th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, MEMICS*. Ed. by Bouda, J.; Holík, L.; Kofron, J.; Strejcek, J.; Rambousek, A. Vol. 233. EPTCS. 2016, pp. 61–71.

[BK11] Beyer, D.; Keremoglu, M. E.: CPAchecker: A Tool for Configurable Software Verification. In: *CAV*. Ed. by Gopalakrishnan, G.; Qadeer, S. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 184–190.

[BKR22] Beyer, D.; Kanav, S.; Richter, C.: Construction of Verifier Combinations Based on Off-the-Shelf Verifiers. In: *Fundamental Approaches to Software Engineering -FASE 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*. Ed. by Johnsen, E. B.; Wimmer, M. Vol. 13241. Lecture Notes in Computer Science. Springer, 2022, pp. 49–70.

[BKW10] Beyer, D.; Keremoglu, M. E.; Wendler, P.: Predicate abstraction with adjustable-block encoding. In: *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*. Ed. by Bloem, R.; Sharygina, N. IEEE, 2010, pp. 189–197.

[BLB13] Becker, M.; Luckey, M.; Becker, S.: Performance Analysis of Self-Adaptive Systems for Requirements Validation at Design-Time. In: *Proceedings of the 9th ACM SigSoft International Conference on Quality of Software Architectures (QoSA'13)*. 2013, pp. 43–52

[CH00] Claessen, K.; Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Ed. by Odersky, M.; Wadler, P. ACM, 2000, pp. 268–279.

[CHJW17] Czech, M.; Hüllermeier, E.; Jakobs, M.; Wehrheim, H.: Predicting rankings of software verification tools. In: *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics, SWAN@ESEC/SIGSOFT FSE*. Ed. by Baysal, O.; Menzies, T. ACM, 2017, pp. 23–26.

[CJS$^+$16] Chalupa, M.; Jonás, M.; Slaby, J.; Strejcek, J.; Vitovská, M.: Symbiotic 3: New Slicer and Error-Witness Generation - (Competition Contribution). In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS nProceedings*. Ed. by Chechik, M.; Raskin, J. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 946–949.

[CLM04] Chen, T. Y.; Leung, H.; Mak, I. K.: Adaptive Random Testing. In: *Advances in Computer Science - ASIAN 2004, Higher-Level Decision Making, 9th Asian Computing Science Conference, Proceedings*. Ed. by Maher, M. J. Vol. 3321. Lecture Notes in Computer Science. Springer, 2004, pp. 320–329.

[DPVZ17] Demyanova, Y.; Pani, T.; Veith, H.; Zuleger, F.: Empirical software metrics for benchmarking of verification tools. In: *Formal Methods Syst. Des.* 50 (2017), no. 2-3, pp. 289–316.

[FH10] Fürnkranz, J.; Hüllermeier, E.: Preference Learning and Ranking by Pairwise Comparison. In: *Preference Learning*. Ed. by Fürnkranz, J.; Hüllermeier, E. Springer, 2010, pp. 65–82.

[GBM17] Galhotra, S.; Brun, Y.; Meliou, A.: Fairness testing: testing software for discrimination. In: *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*. Ed. by Bodden, E.; Schäfer, W.; Deursen, A. van; Zisman, A. ACM, 2017, pp. 498–510.

[GMMP09] Grabisch, M.; Marichal, J.; Mesiar, R.; Pap, E.: *Aggregation Functions*. Cambridge University Press, 2009

[HCD+13] Heizmann, M.; Christ, J.; Dietsch, D.; Ermis, E.; Hoenicke, J.; Lindenmann, M.; Nutz, A.; Schilling, C.; Podelski, A.: Ultimate Automizer with SMTInterpol - (Competition Contribution). In: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*. Ed. by Piterman, N.; Smolka, S. A. Vol. 7795. Lecture Notes in Computer Science. Springer, 2013, pp. 641–643.

[HR92] Horwitz, S.; Reps, T. W.: The Use of Program Dependence Graphs in Software Engineering. In: *Proceedings of the 14th International Conference on Software Engineering*. Ed. by Montgomery, T.; Clarke, L. A.; Ghezzi, C. ACM Press, 1992, pp. 392–411.

[LD22] Leeson, W.; Dwyer, M. B.: Graves-CPA: A Graph-Attention Verifier Selector (Competition Contribution). In: *Tools and Algorithms for the Construction and Analysis of SystemsTACAS, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Part II*. Ed. by Fisman, D.; Rosu, G. Vol. 13244. Lecture Notes in Computer Science. Springer, 2022, pp. 440–445.

[MB08] Moura, L. M. de; Bjørner, N. S.: Z3: An Efficient SMT Solver. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS Proceedings*. Ed. by Ramakrishnan, C. R.; Rehof, J. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.

[MH16] Melnikov, V.; Hüllermeier, E.: Learning to Aggregate Using Uninorms. In: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD, Proceedings, Part II*. Ed. by Frasconi, P.; Landwehr, N.; Manco, G.; Vreeken, J. Vol. 9852. Lecture Notes in Computer Science. Springer, 2016, pp. 756–771.

[MH19] Melnikov, V.; Hüllermeier, E.: Learning to Aggregate: Tackling the Aggregation/Disaggregation Problem for OWA. In: *Proceedings of The 11th Asian Conference on Machine Learning, ACML*. Ed. by Lee, W. S.; Suzuki, T. Vol. 101. Proceedings of Machine Learning Research. PMLR, 2019, pp. 1110–1125.

[MLL04] Moon, S.-i.; Lee, K. H.; Lee, D.: Fuzzy branching temporal logic: Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 34 (2004), no. 2, pp. 1045–1055

[MW15] Mohr, F.; Walther, S.: Template-Based Generation of Semantic Services. In: *ICSR*. Ed. by Schaefer, I.; Stamelos, I. Vol. 8919. Lecture Notes in Computer Science. Springer, 2015, pp. 188–203.

[PTF+21] Pellegrini, G.; Tibo, A.; Frasconi, P.; Passerini, A.; Jaeger, M.: Learning Aggregation Functions. In: *Proc. IJCAI, Thirtieth International Joint Conference on Artificial Intelligence*. 2021

[RHJW20] Richter, C.; Hüllermeier, E.; Jakobs, M.; Wehrheim, H.: Algorithm selection for software validation based on graph kernels. In: *Autom. Softw. Eng.* 27 (2020), no. 1, pp. 153–186.

[RW19] Richter, C.; Wehrheim, H.: PeSCo: Predicting Sequential Combinations of Verifiers - (Competition Contribution). In: *Tools and Algorithms for the Construction and Analysis of Systems TACAS: TOOLympics, Part III*. Ed. by Beyer, D.; Huisman, M.; Kordon, F.; Steffen, B. Vol. 11429. Lecture Notes in Computer Science. Springer, 2019, pp. 229–233.

[RW20] Richter, C.; Wehrheim, H.: Attend and Represent: A Novel View on Algorithm Selection for Software Verification. In: *35th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 2020, pp. 1016–1028.

[SDNW21] Sharma, A.; Demir, C.; Ngomo, A. N.; Wehrheim, H.: MLCHECK- Property-Driven Testing of Machine Learning Classifiers. In: *20th IEEE International Conference on Machine Learning and Applications, ICMLA*. Ed. by Wani, M. A.; Sethi, I. K.; Shi, W.; Qu, G.; Raicu, D. S.; Jin, R. IEEE, 2021, pp. 738–745.

[SMHW22] Sharma, A.; Melnikov, V.; Hüllermeier, E.; Wehrheim, H.: Property-Driven Testing of Black-Box Functions. In: *10th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE*. ACM, 2022, pp. 113–123.

[SS02] Schölkopf, B.; Smola, A. J.: *Learning with Kernels: support vector machines, regularization, optimization, and beyond*. Adaptive computation and machine learning series. MIT Press, 2002.

[SW20] Sharma, A.; Wehrheim, H.: Higher income, larger loan? monotonicity testing of machine learning models. In: *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Ed. by Khurshid, S.; Pasareanu, C. S. ACM, 2020, pp. 200–210.

[TKK+14] Tulsian, V.; Kanade, A.; Kumar, R.; Lal, A.; Nori, A. V.: MUX: algorithm selection for software model checkers. In: *11th Working Conference on Mining Software Repositories, MSR Proceedings*. Ed. by Devanbu, P. T.; Kim, S.; Pinzger, M. ACM, 2014, pp. 132–141.

[UAC18] Udeshi, S.; Arora, P.; Chattopadhyay, S.: Automated directed fairness testing. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE*. Ed. by Huchard, M.; Kästner, C.; Fraser, G. ACM, 2018, pp. 98–108.

[WW13] Walther, S.; Wehrheim, H.: Knowledge-Based Verification of Service Compositions - An SMT Approach. In: *2013 18th International Conference on Engineering of Complex Computer Systems*. IEEE Computer Society, 2013, pp. 24–32.

[WW14] Walther, S.; Wehrheim, H.: Verified Service Compositions by Template-Based Construction. In: *FACS*. Ed. by Lanese, I.; Madelaine, E. Vol. 8997. Lecture Notes in Computer Science. Springer, 2014, pp. 31–48.

[WW16] Walther, S.; Wehrheim, H.: On-the-fly construction of provably correct service compositions - templates and proofs. In: *Sci. Comput. Program.* 127 (2016), pp. 2–23.

[ZHML22] Zhang, J. M.; Harman, M.; Ma, L.; Liu, Y.: Machine Learning Testing: Survey, Landscapes and Horizons. In: *IEEE Trans. Software Eng.* 48 (2022), no. 2, pp. 1–36.

[ZWS+20] Zhang, P.; Wang, J.; Sun, J.; Dong, G.; Wang, X.; Wang, X.; Dong, J. S.; Dai, T.: White-box fairness testing through adversarial sampling. In: *ICSE '20: 42nd International Conference on Software Engineering*. Ed. by Rothermel, G.; Bae, D. ACM, 2020, pp. 949–960.