Quantifying Software Correctness by Combining Architecture Modeling and Formal Program Analysis

Florian Lanzinger
Karlsruhe Institute of Technology
Karlsruhe, Germany
lanzinger@kit.edu

Christian Martin
Karlsruhe Institute of Technology
Karlsruhe, Germany
christian.martin@kit.edu

Frederik Reiche
Karlsruhe Institute of Technology
Karlsruhe, Germany
frederik.reiche@kit.edu

Samuel Teuber
Karlsruhe Institute of Technology
Karlsruhe, Germany
teuber@kit.edu

Robert Heinrich
Karlsruhe Institute of Technology
Karlsruhe, Germany
robert.heinrich@kit.edu

Alexander Weigl
Karlsruhe Institute of Technology
Karlsruhe, Germany
weigl@kit.edu

ABSTRACT

Most formal methods see the correctness of a software system as a binary decision. However, proving the correctness of complex systems completely is difficult because they are composed of multiple components, usage scenarios, and environments. We present QuAC, a modular approach for quantifying the correctness of service-oriented software systems by combining software architecture modeling with deductive verification. Our approach is based on a model of the service-oriented architecture and the probabilistic usage scenarios of the system. The correctness of a single service is approximated by a coverage region, which is a formula describing which inputs for that service are proven to not lead to an erroneous execution. The coverage regions can be determined by a combination of various analyses, e.g., formal verification, expert estimations, or testing. The coverage regions and the software model are then combined into a probabilistic program. From this, we can compute the probability that under a given usage profile no service is called outside its coverage region. We also present an implementation of QuAC for Java using the modeling tool Palladio and the deductive verification tool KeY. We demonstrate its usability by applying it to a software simulation of an energy system.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**; **Object oriented architectures**.

KEYWORDS

Service-oriented architecture, Component-based architecture, Architecture modeling, Deductive verification, Quantitative verification, Architecture simulation, Software reliability estimation

ACM Reference Format:
Florian Lanzinger, Christian Martin, Frederik Reiche, Samuel Teuber, Robert Heinrich, and Alexander Weigl. 2024. Quantifying Software Correctness by Combining Architecture Modeling and Formal Program Analysis. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605098.3636008

SAC '24, April 8–12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04.
https://doi.org/10.1145/3605098.3636008

1 INTRODUCTION

Motivation. Vehicles and critical infrastructure are increasingly governed by software. Therefore, verifying that software behaves according to its specification is becoming ever more important. However, most formal methods used to prove a program's correctness simply output a binary decision, which does not do justice to large software systems which combine many different components in a complex usage environment.

An individual component may behave provably correctly under assumptions that are not ensured by its environment or, conversely, a component may behave incorrectly for inputs that never appear when used within the evaluated system. Both cases can also be considered more gradually: It may be the case that inputs leading to incorrect behavior are very rare or extremely frequent. This demonstrates the necessity for analyses which reach across multiple components and for more gradual, quantitative assessments. There are some non-binary measures of software reliability based on source code in the literature like test coverage. However, such approaches often do not provide guarantees on the system's reliability for concrete usage scenarios. Many quantitative source code analyses only consider individual components, in which case it remains unclear how to combine the metrics. Related work in the architectural domain, such as by Brosch et al. [6], computes reliability measures based on concrete usage scenarios, the system structure, and abstract behavioral specifications. However, the reliability-relevant information is only estimated by an expert.

Our work proposes to couple a quantitative analysis of the architecture model with a formal analysis of source code. By combining these analyses, we can reason about larger systems while still providing fine-grained, quantitative feedback on reliability. In addition, the coupling enables the usage of implementation details in the architectural analysis rather than relying on assumptions by an software architect [23].

Contribution. We introduce QuAC ("*Qu*antifying *A*rchitecture and *C*ode"), an approach to quantitatively determine the probability that a piece of software respects its contract by combining architecture modeling and formal source-code analysis. QuAC answers the

following query: *"With what probability does a typical usage of the system not lead to an error?"* By *error*, we mean any program state contrary to a method's contract. Specifically, an error occurs whenever a method terminates in a state violating its postcondition and whenever it calls another method in a state violating that method's precondition. To describe these states, we use *coverage regions* for every method: a coverage region is a formula describing inputs for which the current method is proven to behave correctly. All coverage regions are inserted into an architectural model. Based on this model and a usage profile describing the typical system usage, we use model counting to calculate a *coverage probability*, which is an under-approximation of the *correctness probability* that the overall program behaves correctly.

To find coverage regions, we consider several approaches, which makes QUAC applicable both in the early and late stages of software development. In the early stages, they are estimated by the developers. These estimations are later replaced by testing and verification results, which are the main approaches considered in this paper. In operation, the usage profile and coverage regions can be further updated by information monitored at run time. This makes QUAC incremental: Information gathered in different phases leads to more precise modeling and thus to a better risk assessment. In addition, it allows developers to combine static verification approaches, which are powerful but expensive – especially when aiming for 100% proof coverage – with run-time verification and testing approaches in a rigorous way.

We formally introduce the QUAC approach and show that it is sound, i.e., it never overestimates the correctness probability. We provide a concrete implementation using the Palladio Component Model (PCM) [22], a metamodel for component-based software architectures and usage profiles, KeY [1], an interactive theorem prover, and several probabilistic model counters. Finally, we demonstrate its usability in a case study.

Limitations. QUAC can be applied to programs modeled as a service-oriented, component-based architecture. We require an association between source-code elements (methods and classes) and architectural elements (services and components). In addition, the logic in which the coverage regions are expressed is restricted by the analysis tools (e.g., KeY) and the model counter. So far, QUAC can be used for safety properties, i.e., the probability of certain errors given certain input distributions. It is not suitable for other properties like liveness. The implementation only supports synchronous calls and the execution of a single usage profile. It does not support multi-agent analyses. It is also limited to programs without unbounded loops or recursion. We plan to extend it to allow support for more general control flow structures both in the service models and in the extraction of coverage regions.

2 PRELIMINARIES

Architecture and code. We combine fine-grained analyses on the source-code level with coarse-grained analyses on the architectural level. On the source-code level, we have an object-oriented program consisting of *classes* which provide *methods*. On the architectural level, we have a service-oriented, component-based architecture consisting of *components* which provide *services*. We assume a mapping that maps every component to a class and every service to a method (not necessarily vice versa). Component-based behavioral specifications provide, in general, no information about the inner state of a component, i.e., state variables. Every service or method also has a contract consisting of two formulas $(pre, post)$, where *pre* is the precondition, which must hold whenever the service is called, and *post* is the postcondition, which must hold whenever the service terminates. In addition to the structure of the software, we also have *behavioral specifications*, which are rough approximations of the internal behavior of a given service. Such specifications include, for example, control flow constructs, like branches or loops, and calls to other services. We assume that every behavioral specification in the architecture is respected by the implementation of the corresponding method. This consistency can be achieved manually, by applying consistency preservation approaches like the one by Monschein et al. [21] or Vitruvius [17], or by inferring the behavioral specifications from the source code.

Software modeling with the Palladio Component Model. We build upon service-oriented, component-based architectures, which can, e.g., be modeled in the Palladio Component Model (PCM) [22]. In the PCM, architectures consist of independent components, which provide services to the user or to other components. A component must declare all services from other components that it requires. Two components are connected if one component provides a service required by the other one. Therefore, a service can be called from outside a component; it takes a specified number of parameters and returns a result. Service calls are always synchronous. For every service, a behavioral specification is provided in form of Service Effect Specifications (SEFFs). A SEFF contains, among others, nodes for control structures like loops and branches, the usage of parameters in the behavior, external call actions modeling calls to other services and the description of results from external calls and their processing. Conditions in control-flow structures are defined by expressions over values provided by local variables, the parameters and other arbitrary specifications. This SEFF is an abstraction of the source code that actually implements the service: It exactly specifies the service's behavior w.r.t. calls to other services and modifications of the component state, but the implementation may contain additional computations or optimizations not represented in the SEFF. We also use the PCM's usage profile, which describes the usage scenario of the system in form of a flow chart and calls of the public services. It also specifies probabilistic values to model the unknown user input. In essence, usage profiles capture the observation of how the system is or should be used. These probabilistic values are provided either from an expert of the systems domain from prior projects (educated guess) or by measurements [22]. Measurements can be taken from prior versions of the product in an evolution scenario or in early versions taken from a test group. The described system behavior under a usage profile is a computation tree of service calls assembled by the execution of the usage profile and the directly or indirectly called SEFFs and the applied control structures. Because of the probabilistic usage profile, each computation path in the tree has an associated probability.

Source code analysis. The most important tool we use to analyze the source code is KeY, a deductive verification tool used to prove Java programs specified in the Java Modeling Language (JML) [20]. KeY is semi-automatic, meaning that most JML specifications can

be proven automatically, but for more complex cases, the proof can be manually inspected and guided by a human verifier.

We define the necessary concepts of source code analysis through the lens of dynamic logic (DL). A detailed overview of DL is given by Harel et al. [12, 13]. Generic first-order DL is a multi-modal logic which extends first-order logic (FOL) with programs that describe possible state transitions. To assert that some postcondition *post* holds after execution of a program α, we use the box modality: $[\alpha]\, post$. Using the usual logic operators from FOL, we can then specify contracts, such as the following, which asserts that, if the variable x starts with the value 42, then it will always be larger than 42 after we run the program in brackets for any positive a: $x = 42 \land a > 0 \rightarrow [x := x + a]\, x > 42$. Formulas in DL are evaluated using Kripke structures with state transitions where each state $\sigma \in \mathcal{S}$ contains (among other things) a first-order logic structure assigning each variable a value of its respective domain. We use $\sigma \vDash \rho$ to denote that a DL formula ρ holds in $\sigma \in \mathcal{S}$ and $\vDash \rho$ to denote that a formula is valid, i.e., holds in all states.

There are many instances of DL, including Java Dynamic Logic (JavaDL) [1, Ch. 3], which is implemented in KeY to deductively verify Java programs specified by JML contracts. KeY's calculus operates on *sequents* of the form $\phi \Longrightarrow \psi$, where ϕ is the *antecedent* and consists of n formulas ϕ_i while ψ is the *succedent* and consists of m formulas ψ_j. A sequent is satisfied by a state $\sigma \in \mathcal{S}$ iff $\sigma \vDash \bigwedge_i \phi_i \rightarrow \bigvee_j \psi_j$. A sequent satisfied by all states is called valid. A formula ρ can be proven true by applying axioms and rules that construct a proof tree: The root is the sequent $\Longrightarrow \rho$, where the succedent contains the formula to prove and the antecedent is empty. By applying a rule to a node C, we obtain several child nodes $P_1, \ldots, P_n$ such that the validity of all sequents P_i together implies the validity of the sequent C. Rules may be *locally* or *globally sound* [4]: A rule is *globally sound* if the validity of all P_i implies the validity of C and *locally sound* if every state satisfying all P_i also satisfies C. The tree's leaves are called goals and may be closed – i.e., tautologies – or open – i.e., yet to be proven.

We have defined an error as occurring whenever a method terminates without satisfying its postcondition or whenever it calls another method while violating that method's precondition. To handle the second case and to make our analysis modular, we use a contract rule: a sequent like $\phi \Longrightarrow [\text{foo}()]\psi$ is only provable through the two premises $\phi \Longrightarrow pre$ and $\phi, post \Longrightarrow \psi$ where $(pre, post)$ is foo's contract.

In addition to potentially slow, but powerful formal verification tools like KeY, we can also use methods like testing, monitoring or expert estimates. While such methods may be faster and give a good first approximation, their use in QUAC is generally unsound.

3 THEORETICAL OVERVIEW OF QUAC

Figure 1 illustrates our approach. The user-defined architectural model is in the center, defining the software *components* and *services*. Every service is modeled by a behavioral specification. In addition, the component model contains a usage profile, which gives a distribution for the initial state and tells us what services are called in the usage of the system. On the left, we have the source code, which we assume is consistent with the behavioral specifications, and which includes a contract for every method. After calculating coverage regions, we add them to the service models. The extended service models and the usage profile are then translated into a probabilistic model, from which a model checker computes the *coverage probability*, i.e., the probability that no service is ever called with an input outside its coverage region. If all coverage regions are correct, the coverage probability is less than or equal to the *correctness probability*, i.e., the probability that the implementation never violates its specification.

Sec. 3.1 defines the parts of our architecture model in more detail. Sec. 3.2 then explains how we compute the coverage probability from the architecture model on a theoretical level and proves that the coverage probability always under-approximates the correctness probability. Sec. 4 explains how coverage regions are computed using deductive verification or testing. Sec. 5 explains how the computation of the coverage probability is implemented.

3.1 Modeling the System

Modeling a service. We model a service by two parts: its *coverage region*, a formula describing the inputs for which its implementation is proven to behave correctly, and its *behavioral specification*, which is an abstraction of the implementation describing how the service modifies the component state and what other services it calls. In addition, we assume that every service has a contract (*pre*,*post*).

Definition 3.1 (Service model). Given a service s, let Σ be a signature which includes the parameters of s and every component's state variables. Let Fml_Σ be the set of all quantifier-free first-order formulas and $Term_\Sigma$ the set of all terms over Σ.

A service model is a tuple (cov, beh), where $cov \in Fml_\Sigma$ is a coverage region and *beh* is a sequence of statements: every statement is either a conditional assignment `if` (*fml*) *var* = *trm* assigning the value of $trm \in Term_\Sigma$ to a variable $var \in \Sigma$ of the same type if $fml \in Fml_\Sigma$ holds, a conditional service call `if` (*fml*) *var* = $f(trm_1, \ldots, trm_n)$ calling the service f with the well-typed parameters trm_i and assigning the result to *var* if *fml* holds, or a conditional termination statement `if` (*fml*) $\lightning$ that terminates the service prematurely.

Coverage regions. A coverage region $cov \in Fml_\Sigma$ describes the input parameters and component state under which a service's implementation is proven to behave correctly, i.e., not terminate in a state that violates the postcondition nor call another method in a state that violates the callee's precondition. It is determined by validating the method that implements the service separately from all other methods using testing, formal methods, or run-time monitoring. For services which are not yet implemented in an early development stage, the coverage region can also be estimated to obtain a what-if analysis. A coverage region does not necessarily represent the set of initial states for which the service behaves correctly, but the possibly smaller set of initial states for which we can prove that this is the case. For the soundness in Thm. 3.6, we must require that every coverage region be correct, i.e., that no input inside the coverage region lead to an error.

Definition 3.2 (Errors and correctness regions). For a given method f and its postcondition *post*, an error is either of the following: a terminal program state for f in which *post* does not hold, or a

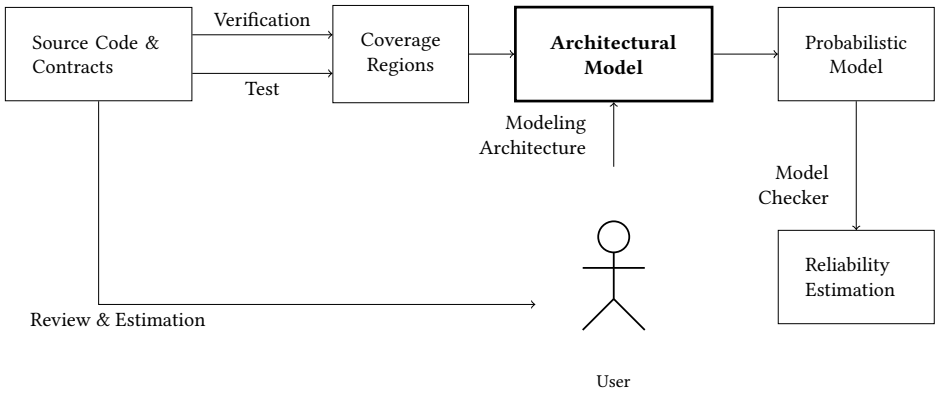


Figure 1: An overview of the QuAC workflow.

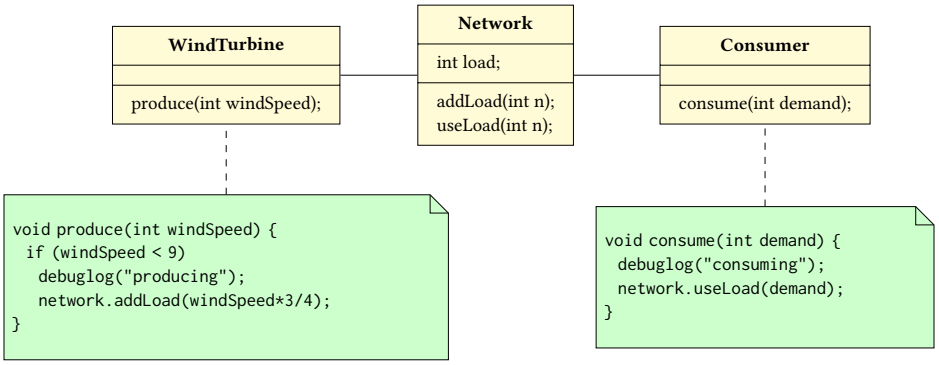


Figure 2: An example software architecture and implementation.

program state in which f calls another method g with precondition *pre* in which *pre* does not hold.

We define the method's correctness region *correct* to be the formula which holds exactly in those initial states in which executing f does not lead to an error.

Definition 3.3 (Correct coverage regions). For a given method f and its contract $(pre, post)$, a coverage region *cov* is correct iff $cov \rightarrow correct$.

Since a coverage region describes the inputs for which a service's implementation is proven to behave correctly, any coverage region computed by a sound proof calculus is correct. While any correct coverage region suffices to make our approach sound, larger regions (i.e., weaker formulas) make it less precise by decreasing the coverage probability. The goal is thus to find maximal correct coverage regions.

Behavioral specifications. The behavioral specification *beh* of a service model *s* is a sequence of statements according to Def. 3.1.

The implementation may be more complex than *beh* and contain additional statements not represented in *beh*. However, the implementation's behavior w.r.t. calls to other services and changes to state variables must be specified exactly. This is because, e.g., a spurious call in a service model may both decrease (if it leads to an error) and increase (if it changes the component state in way that prevents a future error) the coverage probability.

This requirement is achievable with existing methodologies: In model-driven development, behavioral specifications are created first and then source code is generated from them, being kept consistent with the specification by some consistency preservation approach. However, one can also go the other way around and infer behavioral specifications from source code.

Example. From now, we will use the architecture and source code from Fig. 2 as a running example. It implements a heavily simplified simulation of an energy network, consisting of three components that offer four services in total. The produce() service produces electricity based on the current wind speed; if the wind speed is too high, the service produces no electricity. The consume() service consumes electricity. And the addLoad() and useLoad() services are called by the other two services to modify the current network load. We assume that addLoad() and useLoad() are both specified by the contract $pre = post = (\text{load} \geq 0)$ and the other two services by $pre = (\text{windSpeed} \geq 0)$, $post = true$ and $pre = (\text{demand} \geq 0)$, $post = true$ respectively. In other words, an error occurs if the network load falls below zero.

Since the source code is a refinement of the architecture, it may contain elements not present in the architecture. In our example, the calls to the debug log are present in the source code, but are not a part of the behavioral specifications.

The maximal coverage region for useLoad() is $n \leq$ load. For all other services, the maximal coverage region is *true*. This holds even though consume() may indirectly cause an error by calling useLoad() because coverage regions only consider errors caused by the service directly.

3.2 Approximating the Correctness Probability

Using the parts of the architecture model introduced in the previous subsection, we can compute the coverage probability $1 - Pr(\lightning \mid [\![\mathcal{U}(S)]\!])$ that executing the service models S under the usage profile $\mathcal{U}$ does not lead to a premature termination. If all coverage regions in S are correct, this is less than or equal to the system's correctness probability.

Executable Semantics. We start by defining what it means to execute a service model (*cov*, *beh*): If *cov* does not hold, we immediately terminate prematurely. Otherwise, we execute the behavioral specification *beh*. Note that according to Def. 3.1, every service called by *beh* must either terminate prematurely or return to the caller.

To execute a set of service models S, we must also have a usage profile that tells us what services are called and with what values. This profile models how the system's implementation is used or expected to be used. It is similar to a service model, but also includes probabilistic variable assignments to model probabilistic user actions. In addition, the usage profile may only call a service if its precondition holds.

Definition 3.4 (Usage profile). A usage profile $\mathcal{U}$ is a sequence of statements. Every statement is either a conditional assignment, service call, or termination statement as in Def. 3.1, or a conditional probabilistic assignment `if` (*fml*) *var* $\sim$ *dist* where *dist* is a distribution over the domain of the type of *var*. The condition of every conditional service call in $\mathcal{U}$ must imply the service's precondition.

Definition 3.5 (Semantics of usage profile). Let $\mathcal{U}$ be a usage profile and S a set of service models. Then $[\![\mathcal{U}(S)]\!]$ denotes the set of all finite computation traces which are created by executing $\mathcal{U}$ and S.

Each trace $t \in [\![\mathcal{U}(S)]\!]$ has a probability of occurrence $Pr(t)$ which – since the service models are deterministic – is determined solely by the distributions in the probabilistic assignments in $\mathcal{U}$. The coverage probability can then be computed as the probability that a trace randomly chosen according to the distributions in the usage profile does not terminate prematurely, i.e., never calls a service outside its coverage region.

Returning to our example from Fig. 2, we assume a usage profile which calls produce(windSpeed) followed by consume(demand) where windSpeed is an integer uniformly distributed in $[5, 9]$ and demand is an integer uniformly distributed in $[0, 4]$. Then Quac computes a probability of $\frac{4}{5}$, which is equal to the actual correctness probability. This can be seen by considering that of the 25 possible inputs, only the input windSpeed $= 5$, demand $= 4$ and the four inputs windSpeed $= 9$, demand > 0 lead to an error. A smaller coverage region leads to a lower coverage probability. E.g., if we take the coverage region of useLoad() to be *false*, the probability becomes 0.

Soundness. To ensure that the coverage probability under-approximates the correctness probability, it suffices to show that all coverage regions are correct.

Theorem 3.6. *Let $\mathcal{U}$ a usage profile. Let S, S' be sets of service models s.t. for each service s the coverage region in S for s is smaller or equal to the corresponding one in S'. Then $Pr(\lightning \mid [\![\mathcal{U}(S)]\!]) \geq Pr(\lightning \mid [\![\mathcal{U}(S')]\!])$, where $Pr(\lightning \mid [\![\mathcal{U}(S)]\!])$ expresses the probability that by executing the usage profile $\mathcal{U}$ using service models S, we terminate prematurely.*

Proof. The probability $Pr(\lightning \mid [\![\mathcal{U}(S)]\!])$ is the sum of probabilities of every prematurely terminating trace: $\sum_{t \in [\![\mathcal{U}(S)]\!]} Pr(\lightning \mid t)\, Pr(t \mid [\![\mathcal{U}(S)]\!])$. Note that the service models are deterministic and the only probabilistic choices are in the usage model $\mathcal{U}$. Thus, there is an isomorphism $[\![\mathcal{U}(S)]\!] \simeq [\![\mathcal{U}(S')]\!]$. Assume $t \in [\![\mathcal{U}(S)]\!]$ and a corresponding $t' \in [\![\mathcal{U}(S')]\!]$. For a single t, the probability $Pr(\lightning \mid t)$ is either 0 or 1. Thus, we need to show that if t' terminates prematurely, then so does t. Due to the assumption that each coverage region in S is smaller than or equal to the corresponding one in S', we know that if t' hits calls a service s outside its coverage region, then either the same service s is also called outside its coverage region in t, or t already terminates before s is called. □

This theorem tells us that for any service model S consisting only of correct coverage regions, $Pr(\lightning \mid [\![\mathcal{U}(S)]\!])$ over-approximates the actual error probability, and thus $1 - Pr(\lightning \mid [\![\mathcal{U}(S)]\!])$ under-approximates the actual correctness probability of the implementation.

Guiding with quantitative values. To apply Quac, we must specify and verify all components and all services they provide. However, formally specifying and verifying every service is very labor-intensive. Instead, we can sacrifice soundness for practicability by using Quac to calculate the probability of a certain service being called and ignoring all rarely-called services in our analysis.

Furthermore, the severity of different kinds of errors can be considered: For example, logging is pervasive; each service may invoke the logging service to trace the data processing. But an error during logging may be uncritical if it does not influence other parts of the program. To address this, we can manually set the coverage regions of uninteresting services to *true* or even remove these service from the model entirely (like we did with the debuglog() in our running example) to focus only on those errors which interest us. Alternatively, we can extend the architecture model with a new attribute which reflects the cost of an error in a service and instrument the probabilistic model generated by Quac such that it approximates the expected error cost instead of the correctness probability.

4 COMPUTATION OF COVERAGE REGIONS

This section explains how coverage regions can be computed. As mentioned before, we can use verification tools like KeY or testing approaches, which, while faster, give incorrect results.

Using formal verification to compute coverage regions. Consider a JML contract (*pre*, *post*). Beckert et al. [4] introduce the notion

of *state space coverage*. Given a partially open proof, this is the set of initial states for which we know that the method will satisfy its postcondition. Assuming only locally sound proof rules were applied, this set consists of all entry state in which all open goals hold. Thus, the set of open goals induces a coverage region, as formalized by the following theorem and corollary.

Theorem 4.1 (Local Contract Satisfaction). *Let $\rho = pre \rightarrow [s]$ post be a JavaDL proof obligation for some contract. Let $(\mathscr{O}, \mathscr{C})$ be the open and closed goals of an unfinished proof produced through locally sound rules. If a state σ satisfies all open goals, i.e., $\sigma \vDash \bigwedge_{(\phi \Longrightarrow \psi) \in \mathscr{O}} \left(\bigwedge_{i=1}^{n} \phi_i \rightarrow \bigvee_{j=1}^{m} \psi_j \right)$, then $\sigma \vDash \rho$.*

Proof. Remember that for a locally sound proof rule, any state satisfying all premises also satisfies the conclusion. Thus, any state σ which satisfies the conjunction of all open and closed goals $\bigwedge_{(\phi \Longrightarrow \psi) \in \mathscr{O} \cup \mathscr{C}} \left(\bigwedge_{i=1}^{n} \phi_i \rightarrow \bigvee_{j=1}^{m} \psi_j \right)$ satisfies ρ. Since all closed goals are (universally) valid, any state σ which satisfies the conjunction of all open goals only also satisfies ρ. □

This result directly induces a correct coverage region:

Corollary 4.2 (Open Branches as Correct Coverage Region). *Let ρ and $(\mathscr{O}, \mathscr{C})$ be as before. The following formula is an correct coverage region for ρ:*

$$cov \equiv \bigwedge_{(\phi \Longrightarrow \psi) \in \mathscr{O}} \left(\bigvee_{i=1}^{n} \neg \phi_i \vee \bigvee_{j=1}^{m} \psi_j \right).$$

Given such a *cov*, any *cov*′ such that $cov' \rightarrow cov$ is also a correct coverage region. In particular, for a *cov* in conjunctive normal form, we may remove any atoms within a disjunction that contain variables not represented in the architecture model.

Definition 4.3 (Projection). Let $cov \equiv \bigwedge_i \bigvee_j A_{ij}$ be a formula in conjunctive normal form. Let V be a set of variables. The formula obtained by removing all A_{ij} with variables not in V is called the projection of *cov* on V.

To summarize, for any given service, we use KeY to (automatically or manually) find a partial proof for the service's implementation. We then construct a coverage region from the partial proof's open goals. Lastly, we make that coverage region usable in the architecture model by projecting away all variables that exist in the implementation but not the model.

Approximate results. Testing results can be leveraged in two manners: Either the complement of failed tests shapes a possibly incorrect coverage region, or the successful tests shape a correct, but small coverage region. Similarly, results from the monitoring of deployed software components are usable to obtain approximate coverage regions.

For components which have not yet been implemented, neither verification nor testing is possible. In this case, a domain expert must estimate the coverage regions. This is useful as an approximation during development. After the service is implemented, the estimate can be replaced by tests or verification results.

5 FROM COVERAGE REGIONS TO PROBABILITIES

In this section, we describe how the probabilities to enter a coverage region under a given usage profile are calculated. In Sec. 5.1, we define the necessary elements of the Palladio Component Model (PCM) and explain how to integrate the computed coverage regions. In Sec. 5.2, we explain how to use the resulting model to calculate the coverage probability.

5.1 Integration of Coverage Regions in the Architectural Model

Quac builds on usage profiles and elements of a component-based architecture as presented in Sections 2 and 3.

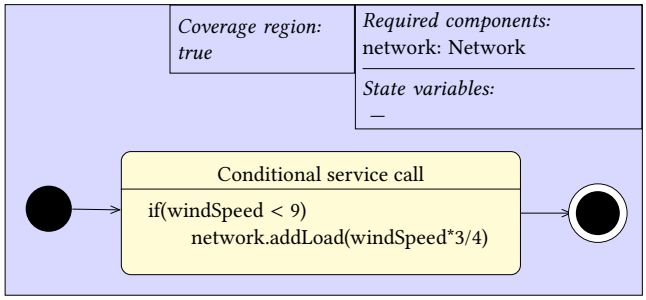
Relevant elements of behavioral specifications for Quac are (1) branching nodes that specify the branching criteria based on the value of a quantifier-free formula over the values of parameters or by a fixed absolute probability. (2) bounded loop nodes that specify the number of iterations based on the values of parameters. (3) external call nodes, which define a call to a service and the provided input for the parameters. These calls may contain a condition over the values of parameters as well as results of other services invocations.

To support Quac's service models in the PCM, it must also model the relevant state of a component using state variables with a name, type, and a probability distribution describing their possible initial values. We introduce state variables and setter and getter nodes in architectural behavioral specifications for the introduction of coverage regions. We also extend all relevant elements which contain some kind of condition to handle values of state variables in addition to parameter values. We extend the PCM with our required information by applying the *inheritance* and *plain referencing* extension approach, described by Heinrich et al. [14]. We provide a separate metamodel with classes extending the existing metaclasses of the PCM, e.g., the getter and setter accesses, or elements which reference an existing element, such as the assignment of state variables to components. With this extension, we can insert the coverage regions as a projection on the available state variables and parameters in the model into the behavioral specification of a service. Conditions for service calls and the symbolic values of the actual arguments can be modeled by developers as part of the behavioral specifications, or computed by KeY in a similar way as the coverage regions. The rest of the behavioral specification can be manually modeled or extracted from the source code with reverse engineering approaches, such as the one by Becker et al. [3].

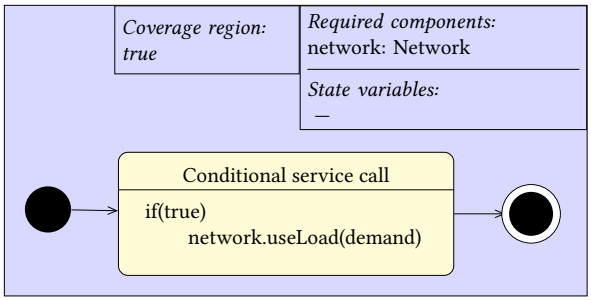
Figure 3 depicts the behavioral specifications corresponding to the services from our running example.

5.2 Calculation of the Probabilities

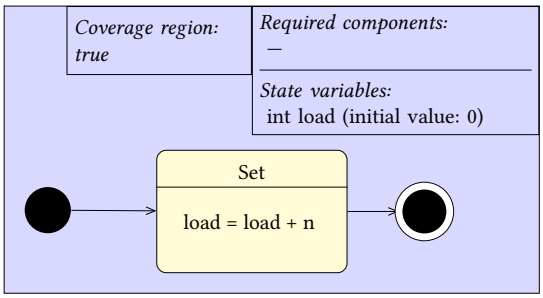
Given a usage profile and the architectural model with the information described in Sec. 5.1, we compute the coverage probability. To this end, we translate each service model into a function in a probabilistic program based on the service model's executable semantics as presented in Sec. 3. The usage profile is translated into the main function with which the execution starts. Calls to services by the usage profile or by other services can be translated into function



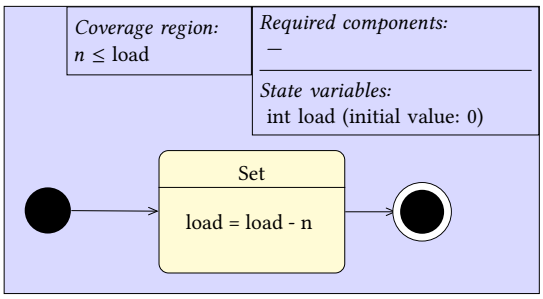
(a) Behavioral specification for produce().



(b) Behavioral specification for consume() .



(c) Behavioral specification for addLoad().



(d) Behavioral specification for useLoad().

Figure 3: Behavioral specifications for the services from Fig. 2.

calls. Branching nodes and bounded loops are translated to `if` or `while` statements in the functions with a corresponding condition.

For example, let us again consider the behavioral specifications in Figure 3. We assume the same usage profile given in Sec. 3. Then the translation into a probabilistic program is shown in Fig. 4.

```
int load;

fun main():
  load = 0; windSpeed ~ U(5,9); demand ~ U(0,4);
  if (windSpeed >= 0): produce(windSpeed));
  if (demand >= 0): consume(demand);

fun produce(int windSpeed):
  if (!(true)): ↯;
  if (windSpeed < 9): addLoad(windSpeed * 3 / 4);

fun consume(int demand):
  if (!(true)): ↯;
  if (true): useLoad(demand);

fun addLoad(int n):
  if (!(true)): ↯;
  load = load + n;

fun useLoad(int n):
  if (!(n <= load)): ↯;
  load = load - n;
```

Figure 4: Probabilistic program corresponding to Fig. 2.

Now, a model checker or model counter, e.g., PSI [9], Storm [15], or counterSharp [24], can resolve the probabilities in Fig. 4.

5.3 Prototypical implementation

We provide a prototypical implementation of QUAC using KeY as the verification tool and the PCM as an architectural description language [19]. We extend the PCM by introducing state variables, getter and setter actions in the SEFFs as well as elements for the attributes of our service model (Def. 3.1).

We extract the coverage regions with KeY and project the results onto the state variables and parameters present in the model. We can also use KeY to extract conditions of service calls.

We implement a transformation from the extended PCM into a probabilistic program in the PSI [9] language similar to the one described in Sec. 5.2. Transformations to other model checkers are possible, but not yet implemented.

6 CASE STUDY

Structure. To demonstrate the feasibility of QUAC, we first ran our running example through our implementation and analyzed the resulting probabilistic program with PSI. We obtained the expected result of $\frac{4}{5}$ (see Sec. 3). KeY found all coverage regions automatically in 26.3s and PSI computed the probability in 1.3s.[1]

To test QUAC's scalability, we consider a more complex version of the running example, shown in Fig. 5. Instead of one, we have three energy producers: a wind turbine, a photovoltaic system, and a gas turbine. In addition, we have an environment sensor, which supplies the current wind speed and sun irradiance.

[1]On average over 3 trials in our artifact VM with 16GB RAM and 1 CPU.

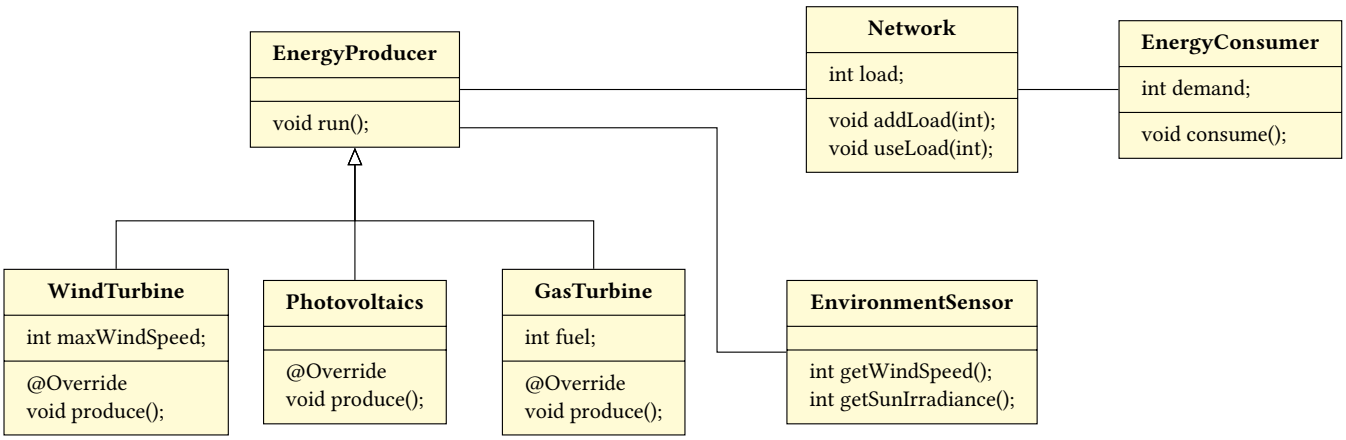


Figure 5: Class diagram for the case study.

```
wt.maxWindSpeed = 15; gt.fuel = 2;

repeat ITERATIONS times:
  cons.demand = D1;
  sens.sunIrradiance = D1;
  sens.windSpeed = D2;

  wt.produce(); phv.produce();
  if netw.load < cons.demand:
    gt.produce();
  cons.consume();
```

Figure 6: Usage profile for the case study.

In our usage profile, shown in Fig. 6, initial values for the demand, sun irradiance, and wind speed are independently chosen from the random distributions D_1, D_2. Then, both the wt.produce() and phv.produce() services are called and increase the network load based on the current wind speed and sun irradiance respectively. If the wind speed exceeds 15, the turbine produces no energy. If the demand is greater than the network load, GasTurbine.produce() may be called as well; this decrements the gas turbine's fuel and increases the network load if there is still fuel left. Lastly, Consumer.consume() is called and lowers the network load. All of this is repeated n times. Residual load is carried over into the next iteration.

We consider the following versions of this usage profile: First, we vary the distributions used. In the first version, $D_1 = \mathscr{U}(0, 1999)$, $D_2 = \mathscr{U}(0, 19)$. In the second version, $D_1 = \mathscr{N}_d(\mu = 1000, \sigma = 32)$, $D_2 = \mathscr{N}_d(\mu = 10, \sigma = 3)$ where $\mathscr{N}_d$ is a discretized version of a normal distribution in which all possible values are integers. For both distributions, we also vary the number of iterations. We consider between 2 to 8 iterations and expect the run time of the model checkers to rise and the coverage probability to sink with the number of iterations.

The system architecture is modeled in the PCM, and there is a SEFF given for every service. The SEFFs offer a behavioral specification for each service, as explained in Sec. 5. In addition, all classes are implemented in Java and specified in JML. Every method implementation is consistent with the corresponding service's SEFF. The JML specification states that the network load is always non-negative.

Extraction of coverage regions. We extract the coverage regions (see Sec. 4) and service call conditions (see Sec. 5.1) from the source code using KeY, then add them to the SEFFs. This yields the coverage region arg ≤ load for Network.useLoad() and the coverage region *true* for all other services.[2]

Computation of the coverage probability. Having computed the coverage regions, we add them to the SEFFs. Then, the usage profile and the extended SEFFs are translated into one probabilistic program, as explained in Sec. 5.2.

In addition, we manually translated the extended PCM into Prism and checked the Prism program with the Storm model checker. We also manually translated the extended PCM into a C program that can be checked using the counterSharp approach, which uses model counting in combination with a bounded model checker to quantify the reliability of C programs as the ratio of failing runs to total runs. As this assumes the initial values to be uniformly distributed, the C program manually converts between a uniform distribution and the distributions shown in our usage profile (Fig. 6). Unlike PSI and Storm, counterSharp uses approximate model counting, meaning that the result is not necessarily exact, but can be found in much less time.

Results. KeY ran in 269.3s.[3] All instances of our experiments with exact model counters PSI and Storm ran out of memory. This is because creating branches for every conditional service call and coverage region resulted in a very large state space, which is further enlarged by our usage profile containing $3n$ random variables with $(2000 \cdot 2000 \cdot 20)^n$ total values for n loop iterations in the usage profile.

[2] Actually, the coverage regions extracted by KeY have additional conditions that deal with non-nullness, exception freedom, etc., but since these do not interest us, we manually remove them before adding the coverage regions to the SEFFs.
[3] On average after 3 trials in our artifact VM.

Cycles	Run time		Coverage probability	
	$\mathcal{U}$	$\mathcal{N}_d$	$\mathcal{U}$	$\mathcal{N}_d$
2	1.7s	6.9s	100%	100%
3	13.5s	10.6s	[99.89%, 99.97%]	100%
4	52.5s	14.6s	[99.64%, 99.89%]	100%
...	...		...	
7	267.8s	96.4s	[97.66%, 99.28%]	100%
8	313.9s	timeout	[96.58%, 98.95%]	timeout

Table 1: Results of our case study with counterSharp: The actual coverage probability lies in the provided interval with 80% probability.

However, using approximate model counting as in the counterSharp approach proved to be practical. As we see in Table 1, although counterSharp shows an exponential run-time increase depending on the number of iterations, its run times are on the low side considering the size of our state space.

We conclude that our approach is feasible and that while exact results can only be achieved for very small programs (like our running example), approximate results with a confidence interval [7, 24] can be found in an acceptable run time even for programs with very large state spaces. The reliability of the intervals can be improved by repeating the computation multiple times.

7 RELATED WORK

Quantitative and probabilistic program analysis. Geldenhuys et al. [10] introduce *probabilistic symbolic execution*, where for every branch during symbolic execution, the branch's probability is calculated via model counting and multiplied with the path condition so far. Thus, one obtains a probability for every path. QuAC is also connected to the field of statistical model checking, which applies statistical methods like Monte-Carlo simulation or hypothesis tests to probabilistic models [28]. Gerrard et al. [11] combine over- and under-approximating verification tools to approximate the reachability condition of a given state, resulting in both lower and upper bounds. QuAC differs from these because it is model-based – i.e., the architectural model we use for the probabilistic analysis is more abstract than the source code – and modular – i.e., we can use different analyses for different services.

Model-based Safety and Reliability Analyses. Different approaches use design models to analyze the safety and reliability of software systems. There are approaches that analyze based on design models and their transformations into other formalisms. Huszerl et al. [16] transform UML state charts into stochastic reward nets (SRNs) to perform a quantitative dependability analysis. Cortelessa et al [8] transform architectural UML models into non-functional models in form of General Stochastic Petri Nets (GSPNs). The generated GSPN is used as an input model for a safety and reliability analysis. The approach by Brosch et al. [6] performs a reliability analysis based on an extension of the PCM which enables the annotation of absolute probabilities for software and hardware elements. These approaches rely on the manual annotation of the reliability information retrieved through estimates, experience or calculation. In contrast, QuAC retrieves reliability-relevant information (the critical regions) by an analysis of source code to compute the probabilities to enter critical regions in the software. This approach connects the architectural view explicitly to the final system and avoids erroneous results due to estimations and inconsistencies between model and implementation.

Work by Töberg et al. [26] as well as Tuma et al [25] is similar to QuAC in that it combines architecture and code by using source-code analyses to verify whether assumptions in an architectural model hold in the implementation. However, these approaches do not modify the architectural model based on the results of the source-code analysis.

Kordon et al. [18] propose a discipline they call *verification-driven engineering*, which is based on using software models for verification. They identify several requirements, e.g., a mapping between model elements and a mathematical framework, which we achieve by mapping Palladio components to JML contracts. They also identify several challenges, like the fact that different model properties must be verified with different tools that are difficult to unify in one model. QuAC offers a partial answer by being able to combine different analyses so long as all of their results can be encoded as a critical region.

Cooperative verification. QuAC is related to the field of cooperative verification. Many approaches exist [2, 5, 27] that allow tools to share proof obligations or partial results so that different components of a program or different properties of the same component can be proven by different tools. Alternatively, partial verification results can be used to guide test cases. QuAC's coverage regions serve both as an exchange format that allows multiple approaches to used in QuAC and as partial results that may guide tests or monitoring. But in addition to allowing tools to cooperate, QuAC computes a correctness measure to quantify how much of the program has been verified by a given combination of tools.

8 CONCLUSION

Summary. QuAC is a modular approach to quantify software correctness by combining architecture modeling and formal verification. We defined QuAC formally and implemented it for a subset of Java programs using the modeling tool Palladio, the formal verification tool KeY, and various model counters. We use KeY to analyze each individual service and extract its critical regions, i.e., the conditions under which it possibly behaves erroneously. These regions are combined with the Palladio Component Model, which includes an architecture model and usage profile, to compute an over-approximate error probability for the overall system. Our case study demonstrates that our approach is feasible. While computing an exact error probability is impractical except for small programs, approximative approaches lead to good run times.

Future work. Our approach allows us to modularly combine different kinds of analyses, but we mainly focused on deductive verification. We would like to investigate other analyses like tests and type checkers and formally establish how they can be used to compute coverage regions. In addition, we want to investigate how we may increase QuAC's precision with tools that over- instead of under-approximate the correctness probability. Furthermore,

this paper only considered reliability properties. We want to apply QuAC to security properties by extending the usage profile to capture attacker capabilities and attack costs.

DATA AVAILABILITY STATEMENT

Our implementation, along with the input and logs for the case study, is available on Zenodo. [19]

ACKNOWLEDGMENTS

We would like to thank the reviewers for their valuable feedback.

This work was supported by funding from the topic Engineering Secure Systems and the pilot program Core Informatics of the Helmholtz Association (HGF), KASTEL Security Research Labs, the research project SofDCar (19S21002) funded by the German Federal Ministry for Economic Affairs and Climate Action, and the German Research Foundation (DFG) under project number 499241390, HE8596/3-1 (FeCoMASS).

REFERENCES

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Lecture Notes in Computer Science, Vol. 10001. Springer. https://doi.org/10.1007/978-3-319-49812-6

[2] Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. 2016. StaRVOOrS — Episode II. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 402–415.

[3] Steffen Becker, Michael Hauck, Mircea Trifu, Klaus Krogmann, and Jan Kofroň. 2010. Reverse Engineering Component Models for Quality Predictions. In *2010 14th European Conference on Software Maintenance and Reengineering*. 194–197. https://doi.org/10.1109/CSMR.2010.34

[4] Bernhard Beckert, Mihai Herda, Stefan Kobischke, and Mattias Ulbrich. 2018. Towards a Notion of Coverage for Incomplete Program-Correctness Proofs. In *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11245)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 53–63. https://doi.org/10.1007/978-3-030-03421-4_4

[5] Dirk Beyer and Sudeep Kanav. 2022. CoVeriTeam: On-Demand Composition of Cooperative Verification Systems. 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 I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 561–579. https://doi.org/10.1007/978-3-030-99524-9_31

[6] Franz Brosch, Heiko Koziolek, Barbora Buhnova, and Ralf Reussner. 2012. Architecture-Based Reliability Prediction with the Palladio Component Model. *IEEE Transactions on Software Engineering* 38, 6 (2012), 1319–1339. https://doi.org/10.1109/TSE.2011.94

[7] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2016. Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

[8] Vittorio Cortellessa, Romina Eramo, and Michele Tucci. 2020. From software architecture to analysis models and back: Model-driven refactoring aimed at availability improvement. *Information and Software Technology* 127 (2020), 106362. https://doi.org/10.1016/j.infsof.2020.106362

[9] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 62–83. https://doi.org/10.1007/978-3-319-41528-4_4

[10] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic Symbolic Execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 166–176. https://doi.org/10.1145/2338965.2336773

[11] Mitchell Gerrard, Mateus Borges, Matthew B. Dwyer, and Antonio Filieri. 2022. Conditional Quantitative Program Analysis. *IEEE Transactions on Software Engineering* 48, 4 (2022), 1212–1227. https://doi.org/10.1109/TSE.2020.3016778

[12] David Harel. 1979. *First-Order Dynamic Logic*. Lecture Notes in Computer Science, Vol. 68. Springer. https://doi.org/10.1007/3-540-09237-4

[13] David Harel, Dexter Kozen, and Jerzy Tiuryn. 2001. Dynamic logic. In *Handbook of philosophical logic*. Springer, 99–217.

[14] Robert Heinrich, Misha Strittmatter, and Ralf Reussner. 2021. A Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis. *IEEE Transactions on Software Engineering* 47, 4 (2021), 775–800. https://doi.org/10.1109/TSE.2019.2903797

[15] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. 2022. The probabilistic model checker Storm. *International Journal on Software Tools for Technology Transfer* 24, 4 (01 Aug 2022), 589–610. https://doi.org/10.1007/s10009-021-00633-z

[16] Gábor Huszerl, István Majzik, András Pataricza, Konstantinos Kosmidis, and Mario Dal Cin. 2002. Quantitative Analysis of UML Statechart Models of Dependable Systems. *Comput. J.* 45, 3 (01 2002), 260–277. https://doi.org/10.1093/comjnl/45.3.260

[17] Heiko Klare, Max E. Kramer, Michael Langhammer, Dominik Werle, Erik Burger, and Ralf Reussner. 2021. Enabling consistency in view-based system development — The Vitruvius approach. *Journal of Systems and Software* 171 (2021), 110815. https://doi.org/10.1016/j.jss.2020.110815

[18] Fabrice Kordon, Jérôme Hugues, and Xavier Renault. 2008. From Model Driven Engineering to Verification Driven Engineering. In *Software Technologies for Embedded and Ubiquitous Systems, 6th IFIP WG 10.2 International Workshop, SEUS 2008, Anacarpi, Capri Island, Italy, October 1-3, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5287)*, Uwe Brinkschulte, Tony Givargis, and Stefano Russo (Eds.). Springer, 381–393. https://doi.org/10.1007/978-3-540-87785-1_34

[19] Florian Lanzinger, Christian Martin, Frederik Reiche, Samuel Teuber, Robert Heinrich, and Alexander Weigl. 2022. *QuAC – Quantifying Software Correctness Using Architecture Modeling and Formal Program Analysis*. https://doi.org/10.5281/zenodo.7473812

[20] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. 2013. *JML Reference Manual*. http://www.eecs.ucf.edu/~leavens/JML//refman/jmlrefman.pdf Revision 2344.

[21] David Monschein, Manar Mazkatli, Robert Heinrich, and Anne Koziolek. 2021. Enabling Consistency between Software Artefacts for Software Adaption and Evolution. In *2021 IEEE 18th International Conference on Software Architecture (ICSA)*. 1–12. https://doi.org/10.1109/ICSA51549.2021.00009

[22] Ralf H. Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann (Eds.). 2016. *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press.

[23] Sophie Schulz, Frederik Reiche, Sebastian Hahner, and Jonas Schiffl. 2022. Continuous Secure Software Development and Analysis. In *Symposium on Software Performance 2021 : Short Paper Proceedings of Symposium on Software Performance, Leipzig, Germany, November 9.-10., 2021. Ed.: D. G. Reichelt, R. Müller, S. Becker, W. Hasselbring, A. v. Hoorn, S. Kounev, A. Koziolek, R. Reussner (CEUR Workshop Proceedings, Vol. 3043)*. RWTH Aachen. 46.23.01; LK 01.

[24] Samuel Teuber and Alexander Weigl. 2021. Quantifying Software Reliability via Model-Counting. In *Quantitative Evaluation of Systems*, Alessandro Abate and Andrea Marin (Eds.). Springer International Publishing, Cham, 59–79. https://doi.org/10.1007/978-3-030-85172-9_4

[25] Katja Tuma, Sven Peldszus, Daniel Strüber, Riccardo Scandariato, and Jan Jürjens. 2023. Checking security compliance between models and code. *Software and Systems Modeling* 22, 1 (2023), 273–296.

[26] Jan-Philipp Töberg, Jonas Schiffl, Frederik Reiche, Bernhard Beckert, Robert Heinrich, and Ralf Reussner. 2022. Modeling and Enforcing Access Control Policies for Smart Contracts. In *2022 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. 38–47. https://doi.org/10.1109/DAPPS55202.2022.00013

[27] Petra van den Bos and Marieke Huisman. 2022. The Integration of Testing and Program Verification - A Position Paper. In *A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 13560)*, Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos (Eds.). Springer, 524–538. https://doi.org/10.1007/978-3-031-15629-8_28

[28] Håkan L. S. Younes, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2006. Numerical vs. statistical probabilistic model checking. *Int. J. Softw. Tools Technol. Transf.* 8, 3 (2006), 216–228. https://doi.org/10.1007/s10009-005-0187-8