Model-driven Quantification of Correctness with Palladio and KeY

Frederik Reiche Jonas Schiffl Alexander Weigl
Robert Heinrich Bernhard Beckert Ralf Reussner

December 2020

Abstract

In this report, we present an approach for the quantification of correctness of service-oriented software systems by combining the modeling tool Palladio and the deductive verification approach KeY.

Our approach uses Palladio for modeling the service-oriented architecture, the usage scenarios of the system (called services) in particular, and the distribution of values for the parameters provided by the users. The correctness of a service is modeled as a Boolean condition. We use Palladio to compute the probability of a service being called with *critical parameters*, i.e. , in a way that its correctness condition is violated. The critical parameters are computed by KeY, a deductive verification tool for Java. The approach is not limited to KeY: Other techniques, such as bug finding (testing, bounded model checking) can be used, as well as other verification tools.

We present two scenarios, which we use as examples to evaluate the feasibility of the approach. Finally, we close with remarks on the extension to security properties. Furthermore, we discuss a possible approach to guide developers to locations of the code that should be verified or secured.

1 Introduction

The assessment of the safety and security of a complex software system can not be a binary decision. There are too many obstacles to give an informed and definite decision: First, not all artifacts of deployed components may be accessible, especially if the software models and source code are provided by a third party. Second, the software sub-systems and also the complete deployed software architecture might be too complex to test, verify and validate as a complete system. Third, a sub-system might be secure and safe under isolated analysis, but this analysis is only valid under assumptions on the use and deployment of the component. These assumptions need to be ensured by the component's environment (other components, databases, file storage, etc.), which might be infeasible. Fourth, software is under a steady pressure to evolve. Adaptions are needed for new use cases, or when bug fixes are necessary. This leads to (partial)

changes within the deployed software system, and sub-systems which might have been safe can become vulnerable.

In this report we investigate how quantitative analysis of safety and security properties can be carried out by using the capabilities of Palladio and KeY. Palladio is a modelling tool for software. In our approach it covers the architectural view of the system, whereas KeY, an interactive theorem prover, covers the analysis on the source code level. We combine both tools to achieve an inference machine for the following query: *"With which probability does a typical usage of the system lead to an error?"*. For this purpose, KeY is used to find under which conditions, defined by parameter values or value-regions, a method provides erroneous results or behaviour. These conditions are then projected into the Palladio model of the system. Based on a usage profile and the system model enriched with the error conditions, the probabilities to reach the critical methods and also provide the parameter values to trigger an error are calculated. We also discuss, how and to what degree this approach can be used to guide developers to parts of the system that can be of more interest for fixing the found errors than others. Furthermore, several enhancements are discussed to increase the capabilities of the approach.

2 Combining Architecture with Program Verification

In this report we present a new approach for continuous evaluation of the current operational risk of a (to be) deployed complex software system. The approach combines an architectural modeling of the software components and the validation of single services.

The single services are validated using testing techniques or formal verification. Furthermore, observations during operation, e. g., monitoring of errors, could be used. These validation results can also be modeled by the software engineer. For example, approach utilizing the models of a software engineer is useful when a service is not yet implemented in an early development stage. The results of the validation are in the shape of a Boolean expression over the input variables of the service. The expression describes the set of critical parameter combinations which have a potentiality for a system crash or the execution of erroneous behaviour. For the analysis, we assume calling a service with an input value of a critical region leads to an error.

The validation results are a local assessment of services, for a global view on the complete composed software system we use structural and behavioral modeling of Palladio's diagrams. For a precise risk assessment, we rely on usage models of the services. A usage model describe a usage scenario of the system from the perspective of a user of the public API. The usage model consists of a probabilistic flow chart which contains service calls, branches and loops. Probabilities are used to describe the distribution of taking branches and the distribution of the values of input parameters of the services.

The approach is applicable in the early stages of software development, even

before the testing and validation phase, as well as during the operation. In the early stages, the usage model and critical parameters is mainly guessed by the developers. These guesses are replaced by testing and verification results before the deployment, and in operation the usage model is updated by monitored information of the running system. This makes the approach incremental: Information that is gathered in different phases, like run-time errors, new tests and verification results, lead to preciser critical regions and to a better risk assessment.

2.1 Modeling with Palladio

Palladio is a modeling tool coming with an own notion of service-driven architecture. In the Palladio Component Model (PCM), a software consists out of independent components, which provides services either to the user or to other components. A service can be considered as a functionality which can be called on a component given the service name and parameters—similar to a method invocation. The PCM provides a greybox perspective on component where only a rough approximation about the internal behavior of a called service can be represented.

We use four different models of Palladio in our approach: (1) In the repository model, we define the available components along with their provided and required services. (2) A system model provides the assembly of several components to one system. The system itself provides services to external users that are delegated to internal components which also provide them. The system model therefore specifies which components of the repository model are used in a system. Furthermore, the system model specifies which components interact with the others by specifying connection between their provided and required interface roles. (3) The usage models describe how the system is used in the sense of which public services are called by users from the outside of the system. The usage model therefore captures the observation how the system were used or is expected to be used. The usage model describes the usage of a system as a sequence of service calls, branches for expressing different behaviour possibilities and loops for modelling repeated calls. The number of loop-iterations and the probabilities for taking a branch are specified by probabilities. Also, possible parameter-values that are used as an input for a service have to be specified for each service call. The values of the parameters of a call to the system services can be described through stochastic expressions. These expressions are mathematical expressions which describe a distribution of a variable. They can contain pre-defined probabilistic distributions, or other variables and constants. (4) The behavior of a service provided by a component is described by an activity diagram, called Service Effect Specification (SEFF). A SEFF allows basic control structures like branches and loops, where the branch- or loop-condition can depend on the parameters of the service. These conditions are stochastic expression.

The Palladio tool provides a reasoning machine, the *DependencySolver*, which determine the entry probability of branches and services calls based for a given usage model. The main task of the reasoning is to combine the usage model with the

SEFFs of the called services by propagating the probabilities of the input variables from the usage model into the SEFFs. With this approach, the *DependencySolver* calculates for each stochastic expression in a SEFF the probabilities. This especially includes stochastic expressions that depend on parameters.

2.2 Verification result of KeY

KeY is a deductive verification system for Java programs. Thus, it enables us to verify (formal) properties, specified in the Java Modeling Language (JML) contracts, of Java programs. A contract mainly consists of a pre- and post-condition.

In our approach, we exploit KeY's reasoning capabilities to infer precise critical regions. These critical regions consist of the conditions of program paths for which the post-condition can not be established. Due to the semi-decidability of first order theorem proving, these program paths might actually be correct, even though a proof could not be found. We denote the critical region with $R_{\lightning} = \mathrm{model}(r_{\lightning})$ which we describe as a formula $r_{\lightning}$ over the input parameters of the service.

$$r_{\lightning} := \neg pre \wedge \bigvee_i cond(path_i)$$

Every model of $r_{\lightning}$ is a potential input parameter combination which could lead to a potential failure of the service.

2.3 Analysis outcome

Given a usage model $\mathcal{U}$, we compute the entry probability of a failure. Semi-formally:

$$P(\lightning \mid \mathcal{U}) = \text{probability that an error occurred}$$

An *error occurs* if we call a service with parameters in its critical region.

3 Scenario: Maturity

In this scenario, we consider a small critical component of a web shop: the determination if the buyer is mature. The age of maturity depends on the country and sometimes also on the state (for U.S.). So the system provides a service, which takes the *year of birth* and the current *country code* and decides whether a given person is mature. Internally, this system contains two sub-components, the first one computes the age given the year, the second computes the maturity given age and country (cf. Fig. 4).

Palladio Models. The repository view in Fig. 1 shows the structural overview of our system: It contains three service descriptions (interfaces) and three components. The `AgeOfMaturity` is the publicly provided service, which requires the services

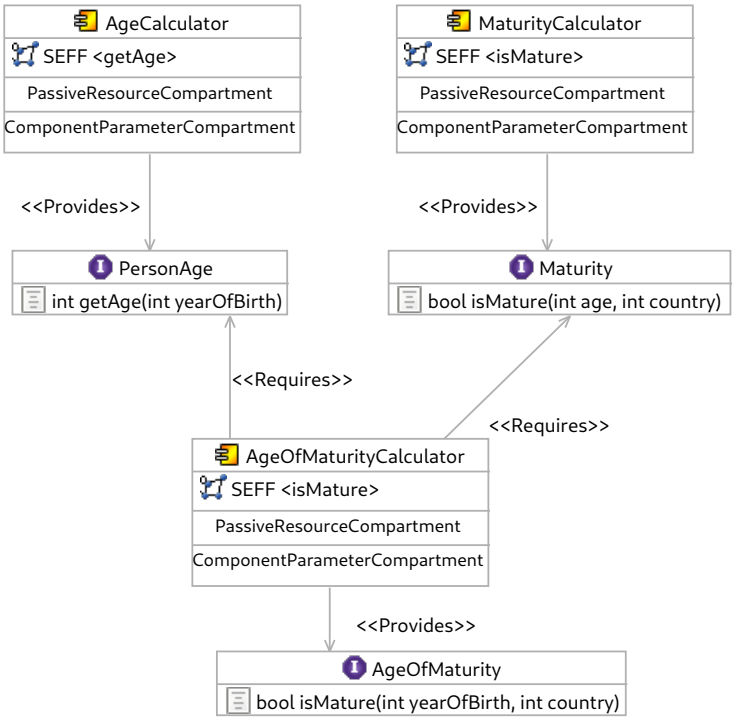

Figure 1: Structural view of the components and implemented services

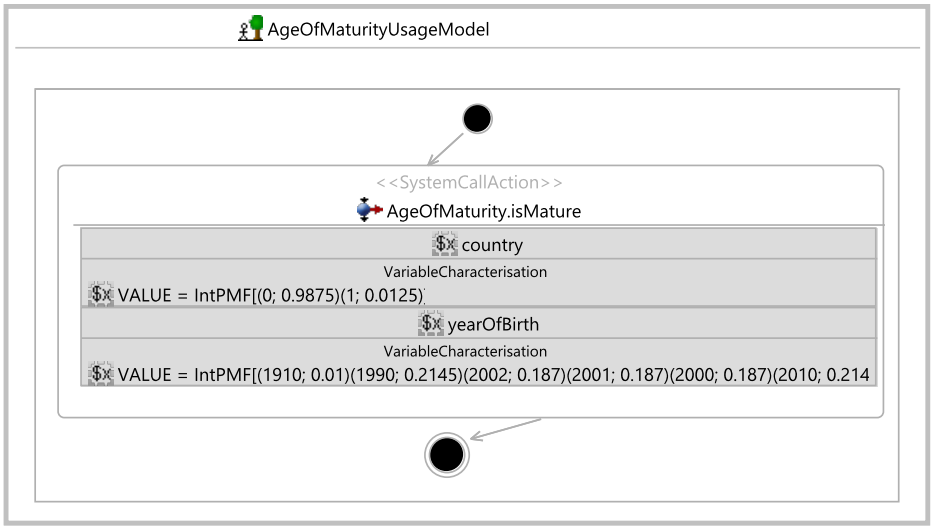

Figure 2: The usage model

$$P(country = EGY) = 0.0125 \qquad P(country \neq EGY) = 0.9875$$
$$P(yearOfBirth = 1910) = 0.01 \qquad P(yearOfBirth = 1990) = 0.2145$$
$$P(yearOfBirth = 2000) = 0.187 \qquad P(yearOfBirth = 2001) = 0.187$$
$$P(yearOfBirth = 2002) = 0.187 \qquad P(yearOfBirth = 2010) = 0.2145$$

Figure 3: The propability distribution for $yearOfBirth$ and $country$

`PersonAge` and `Maturity`. `AgeOfMaturity` is implemented in the component `AgeOfMaturityCalculator`.

In Fig. 2, we present the usage model of our scenario. The usage model contains a simple sequential workflow, describing a single call of the public provided service `AgeOfMaturity`. Moreover, it describes the expected distribution of the parameters. For the analysis we assume following probabilities for the parameter *country* and *yearOfBirth*. Due to limitations of Palladio, it is not possible to define probabilities of value ranges or over any distribution. The probability distribution is a mixture of dirac impulses as seen in Fig. 3.

For any other birth year the probability is zero.

The program code of the service. The implementation of the component `MaturityCalculator` is given in Fig. 4. It is not immediately obvious whether this program is correct.

The verification with KeY reveals the condition for the critical region of this service. The service is incorrectly implemented for Egypt, and it is not able to handle an age over or equal to 100:

$$r_{\lightning}(\texttt{MaturityCalc.}) := age \geq 100 \vee (country = EGY \wedge (18 \leq age \leq 20)) \quad (1)$$

The critical region for the `AgeCalculator` is empty: $r_{\lightning}(\texttt{AgeCalculator}) = false$.

The method contract in Fig. 4 expresses that *country* must be a valid country code (integer) and *age* is greater than 0. And should compute the valid maturity given as the predicate *isMature*. The *assignable* directive forbids a change on the heap.

Modeling the Critical Regions. We need to bring the critical regions of each service into the PCM. For this we modify the SEFFs of the services by adding new branches with guards matching the the critical region. An example for this modification is depicted in Fig. 5. The whole behavior of the SEFF is duplicated in the "error branch" and in the "non error branch". The "error branch" has the exact condition (1), where the "non error branch" contains the negated expression to generate a clear separation. The benefit of encapsulating the whole behavior is, that the probabilities for successive and nesting branches or calls can be calculated

```
/*@ requires AFG <= country && country <= CYP;
  @ requires 0 <= age;
  @ ensures \result <==> isMature(country,age);
  @ assignable \strictly_nothing; */
public boolean isMatureWrong(int country, int age) {
    if( age > 100 ) assert false;
    if( age == 100) throw IllegalArgumentException();
    switch(country) {
        case IRN: return age >= 15;
        case YEM, KGZ, NP, TKM, UZB, VNM:
            return age >= 16;
        case DZA, KOR: return age >= 19;
        case THA, TWN, ARE, NZL,
             JPN:
            return age >= 20;
        case BHR, BDI, CIV, GIN, HND, CMR, LSO,
             NAM, SLE, SGP, SWZ:
            return age >= 21;
        default:
            return age >= 18; } }
```

Figure 4: A **wrong** program code to compute the maturity given in a compressed Java form.

on this branches with the existing tooling. Therefore, a more exact evaluation of the impact of the error on the basis of the occurring probability can be made.

3.1 Results

We successfully applied the *DependencySolver* to compute the probability to reach of the *error branch* $P(r_{\lightning}(\texttt{AgeCalculator}))$. The result is that the critical region is hit with a probability of ca. 0.018, whereas the non-defect case occurs with ca. 0.982, respectively. In this Scenario, because of the assumption that age is equally distributed over all countries, the *DependencySolver* calculated $P(age >= 18) = 1 - 0.7855$, which is the probability independent of the prior probabilities in the call hierarchy. Therefore, the probability of classifying a person wrongfully as mature in this scenario is $P(age >= 18) \times P(r_{\lightning}(\texttt{AgeCalculator})) = 0.014139$.

4 Case Study: Cat Basket for a fair and safe trading of kittens

Trading of animals is a critical business. In this scenario, we look into a blockchain-based service that manages the selling of kittens. The marketplace realized on the blockchain conforms to animal rights where trading kittens is only allowed when

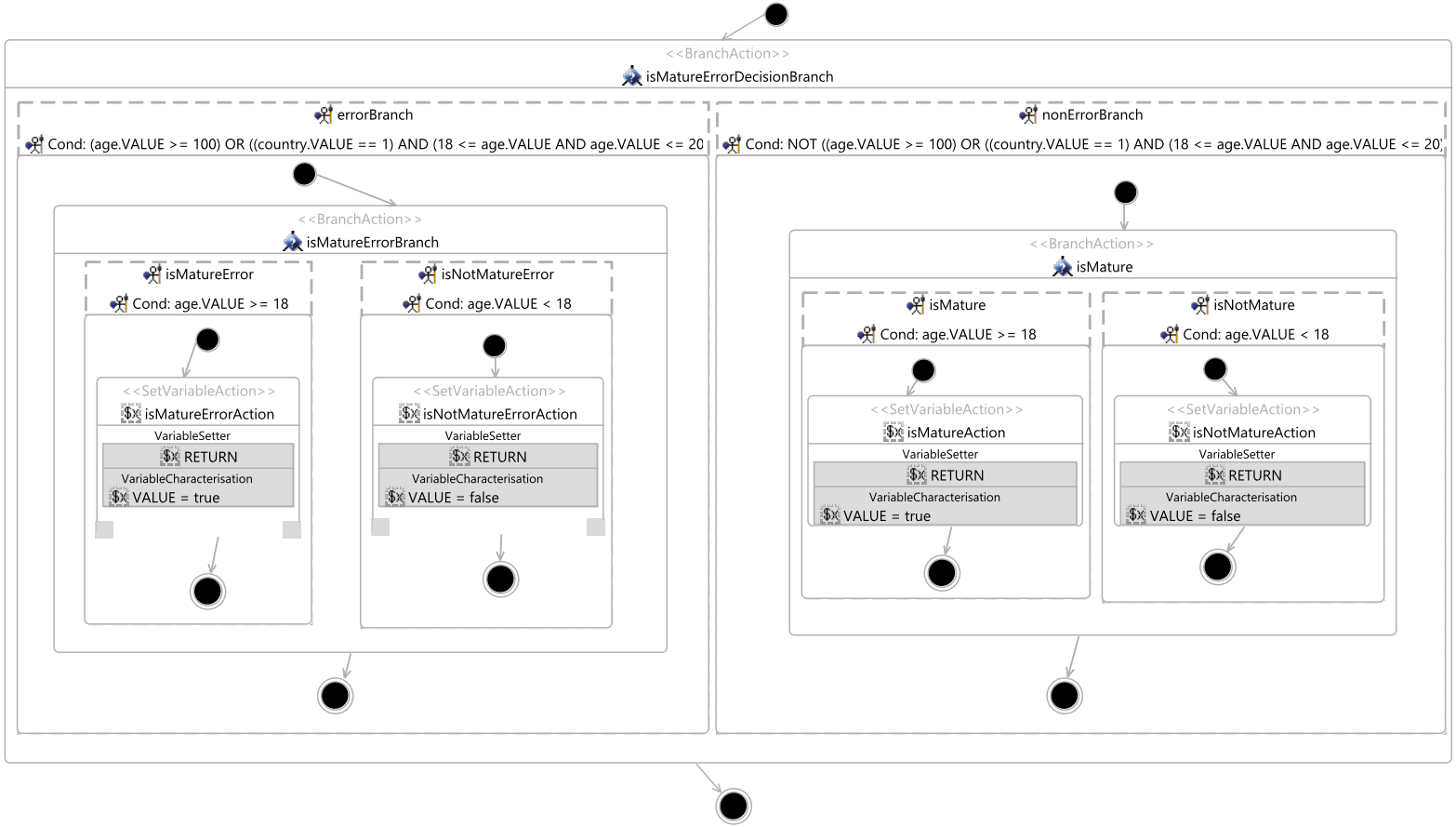

Figure 5: The modeling of the critical regions as SEFF.

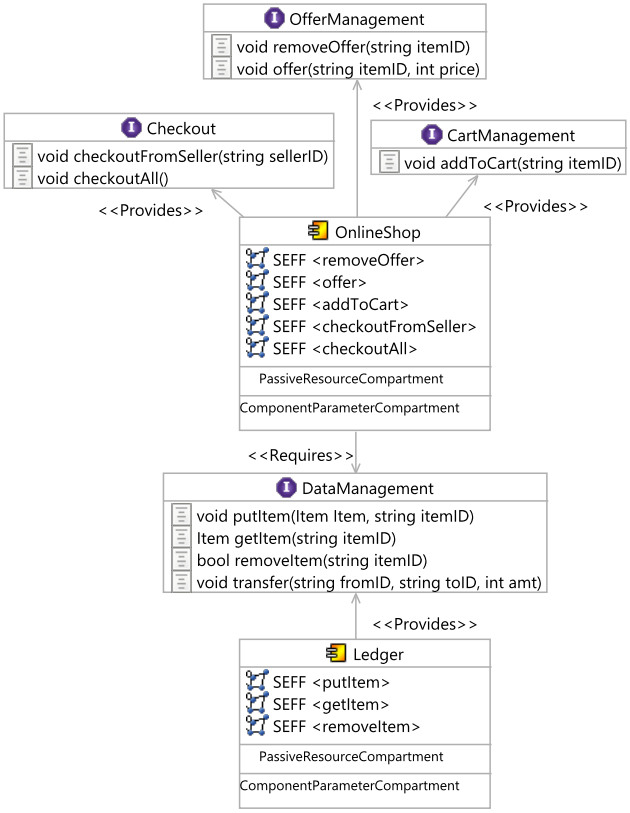

Figure 6: Structural view of the components and implemented services

they are two months old or older. In the meantime, they are offered, and the buyer can reserve an individual kitten. Reservation is associated with placing a large amount of money as a deposit.

This process can be handled by a smart contract, which offers the services of reserving a kitten and placing the deposit. It also handles withdrawal of the deposit, and the final check-out procedure.

In the PCM a smart contract can be modelled with a single component where the necessary operations are provided through interfaces. In fig. 6 the repository view can be seen, where the three interfaces `OfferManagement` (functionality for offer or removing offers of items), `CartManagement` (functionality for reserving offered items) and `Checkout`(functionality for checking out (paying) for items taken by user) exist. The smart contract itself is represented as a single component `OnlineShop` that provides these interfaces. When using Ethereum as the underlying technology, every element of the state is stored as a byte-stream on the blockchain. As an abstraction of this behaviour, the `Ledger` component models the interaction with the blockchain. It provides the interface `LedgerInteraction` which contains operations to transfer ether and to place, retrieve or remove items.

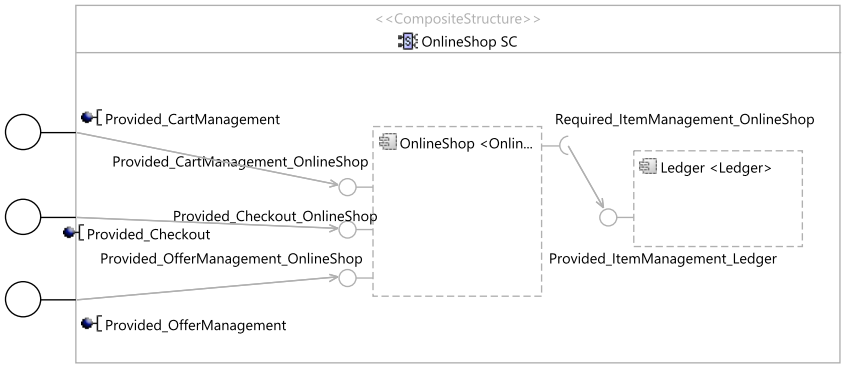

Figure 7: System view of the Cat Basket

The `OnlineShop` components requires the `LedgerInteraction` interface to enable interaction with the blockchain storage. As shown in Fig. 7, the system is represented in the PCM as an `AssemblyContext` containing the `OnlineShop` and `Ledger` components. The system itself provides the same interfaces as the `OnlineShop` component and delegates every call to `OnlineShop`.

The usage model of a possible interaction with the smart contract is shown in Fig. 8. Palladio is focused on business process interactions which involve several interactions between a user and the system. However, in this scenario, every action is executed on the blockchain, and every change is persisted continuously. Thus, most interactions consist only in a single call to the system (e.g. offering an element). Only after an item is reserved for the user (`addToCart`), the user can check out a single or all reserved items or end the interaction with the smart contract. Most smart contracts are simple programs only concerned with enabling the transfer of data when predefined rules hold. In a shopping example, the internal workings of each call therefore remain fairly simple. Because no information by other smart contracts is used, most calls consist of one internal action or, at most, a call to the `Ledger` component for interaction with the blockchain.

5 Quantifying security properties

Our approach is also applicable for security properties. Thus far, the usage models only describe the normal usage of the system by a normal user. For security considerations, we need to model the attackers. For this, we assume that they want to break the system, either by destroying the integrity, availability or confidentiality. Attacks on these goals may include a search for a malicious state of the system and therefore require a series of service calls. All security goals can be modeled (partially) by using critical regions of services. For a detailed analysis, we need to define the capabilities of the considered attackers.

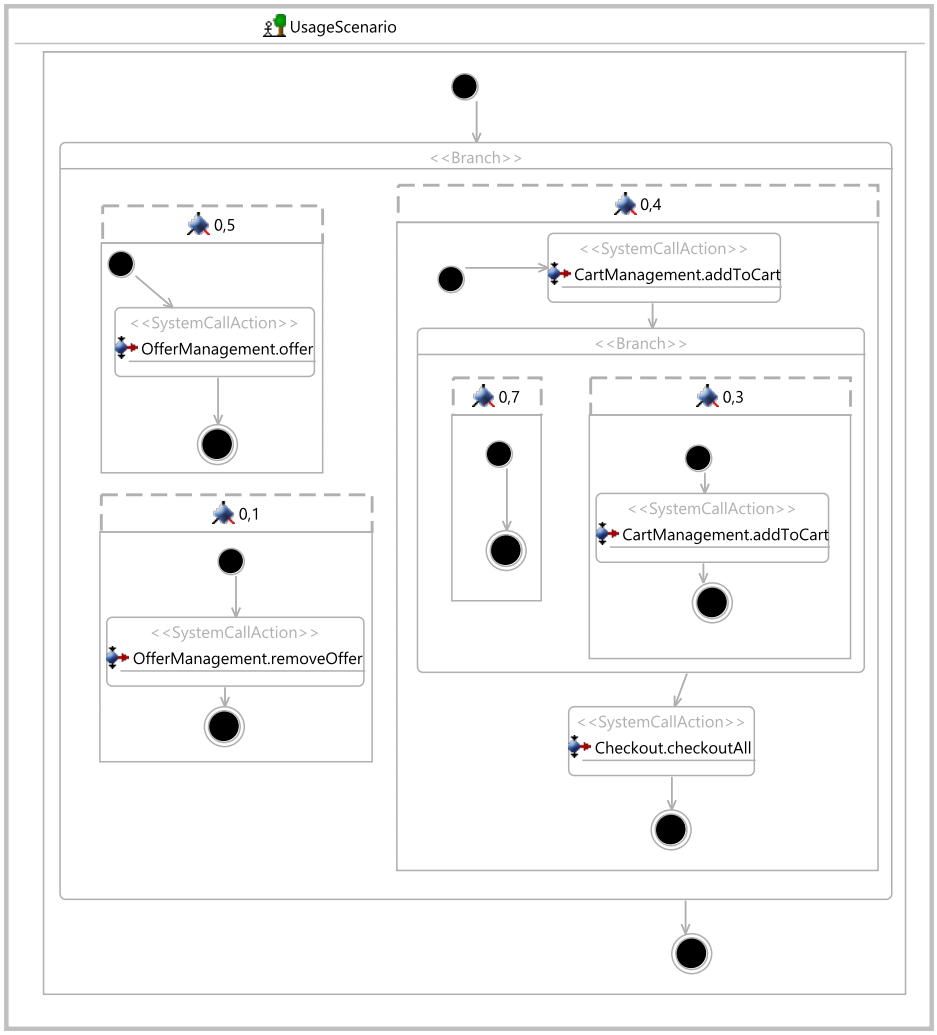

Figure 8: Usage Model of the Cat Basket System

Attacker model. We assume the attackers can apply man-in-the-middle (MITM) attacks between the system and the user. The MITM attack allows the attacker to manipulate the service calls. Thus, they are able to change the invoked service, and the value of the parameters. This also includes that they are able to learn critical parameters, e. g. authentication tokens. On the downside, the attackers cannot manually manipulate the service calls. They need to insert a piece of software that handles the MITM attack. The communication inside the system is considered as trusted and secure, thus the internal service calls can not be manipulated. This limitation can be easily dropped later. The goal of the attackers is to reach a critical region of a service. Hence, they try to manipulate in such a fashion that the error probability is increased.

From usage model to attacker model. The developers and the attackers have something in common: both are not aware of the user behavior. For both the user behavior need to be captured a-priori before the system is deployed. The developers express their knowledge (or guess) of the user behavior into the usage model. The attackers can mimic this procedure with one advantage: Their MITM attack has access to the concrete input values. For the security analysis, the behavior of both (the user and the attacker) need to be modelled. Whereas the behavior of the user is modeled probabilistic, the behavior of the attacker has non-deterministic parts. These parts originate from the a-posteriori knowledge of the concrete parameters. The probabilistic parts models the lack of attacker's knowledge of the system (internal state, other users and their requests) and also origin from a non-deterministic system behavior. Our attacker model does not include the possibility for an attacker to send service requests by themselves (without a normal user request). This implies that an attacker can only change the called service, but is not able to request multiple services for one service call of the user.

Formally, let $\mathcal{U}$ be a usage model which describes the normal system usage, then $\mathcal{A}(\mathcal{U})$ denotes the derived attacker model. The difference between $\mathcal{U}$ and $\mathcal{A}(\mathcal{U})$ is that those service calls and parameters that might be modified by the attackers are tagged. The tag states that a parameter value can be manipulated, or the possibility of redirection to a different service.

For the analysis we want to comprehend the attacker's choice of the manipulation. In particular, the attacker searches for a manipulation which increases the probability of reaching an error $P(\lightning)$. Let A denote a strategy for applying manipulations on given user session. Then formally, the goal of the attacker is to find the strategy A with

$$\max_{A} P_{\mathcal{A}(\mathcal{U})}(\lightning \mid A) \tag{2}$$

Critical regions for modelling the damage potential. To apply our previous analysis, we need to model the damage potential as a safety specification, which should describe a system condition that should be unreachable by the attacker. This

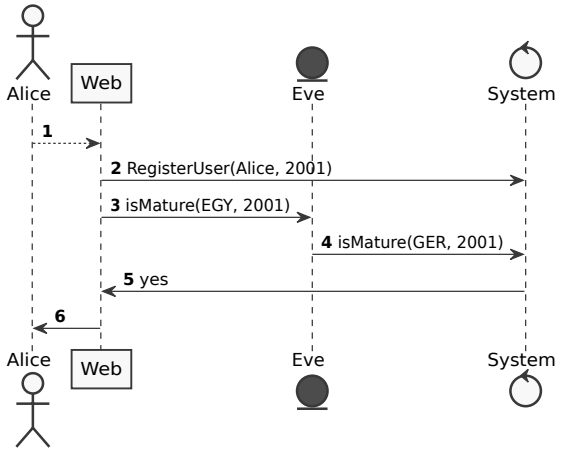

Figure 9: Sequence diagram showing the *Eve*'s attack on *Alice* by hijacking a service call.

means modelling our protection goals as such a safety property. For integrity or availability, this seems easily doable: Integrity describes the validity of the current state. Therefore, it is inherently a match for a safety property. For availability, we may require an additional specification variable to model the load of the system, e.g. the current number of requests. Confidentiality may be hard to describe precisely. Typically, this property is described by a forbidden information flow between secret and public information. Formally, the absence of such a secure information flow is proved via the indistinguishability of two program runs. To cover confidentiality, we need to forbid the program paths that may leak information.

Note that the specification for the critical regions may differ depending on the attacker and its goal. For availability, the attacker tries to provoke an early abortion of the service's execution for all users, typical by an exception. Thus, nobody can use the services in an appropriate fashion. For integrity, the attacker searches for a way to insert malicious information into the system. An abortion of the service execution is obstructive. The same is true for confidentiality, where the attacker tries to steal information. A thrown exception would destroy the publishing of the secret information back to the attacker. Typically, the security specification and functional specification are completely different.

Example. Finally, we want to give an example of what such an analysis could look like. Fig. 9 shows a sequence diagram of a MITM attack for our *Maturity* example. *Eve* is the attacker who hijacks the service request. This request was sent by the web interface in charge of the user *Alice*. The manipulation of the request by *Eve* leads to an incorrect answer of the system, saying that *Alice* is mature. This may lead to penalties.

In this example, we need to mark the parameter `yearOfBirth` in the service

call (in the usage model Fig. 2) as manipulatable by the attacker *Eve*. Manipulatable parameters are under influence of the attacker, who will choose the most fitting value for the attack (reaching the invalid state, or staying undiscovered, etc.).

Let us consider an alternative: At 19 years, *Alice* is of age in Germany. But *Eve* manipulates the country in the request to Egypt. *Alice* is now (for the system) not mature, and probably her next actions would be declined. This attack happens outside of the system: the attacker disturbs the communication, but does not try to bring the system into a bad state. For our analysis, which is based on the critical regions, this attack could not be modelled directly. However, such attacks could be modelled with usage models and do not need any insights of the running system.

6 Developer Guidance with Quantitative Values

For the application of the presented approach, the verification of components and especially their provided methods is necessary. Specification and verification of every method would yield all critical regions. However, this introduces additional effort.

Therefore, quantitative values should be used to guide developers to locations in the system that should be prioritized in the verification. For this endeavor, the probabilities of calling a certain method can be calculated with the *DependencySolver*, by calculating the probabilities of paths through the system. For each path that includes a call to a certain method of a component, the probabilities are multiplied to a total probability for the call to that method. These probabilities can help as a first indicator which methods can be important to verify.

However, the probability of calling certain methods is only a weak metric because the importance of the corresponding code is not included. For instance, in an online-shop a method printing some welcome-text could be called more often than a check-out process. However, verifying the correctness and security of a check-out process methods is more important, due to the processing of critical information like credit card details of the customer. Therefore, for calculating a metric to guide developers to important parts of system that should be verified first, modelling the importance of assets with a quantitative value in the system (e.g. by consulting a security expert) can be considered. For this purpose, an extension of the PCM classifies assets, e.g. parameters, into classes. Each of these classes then gets a value assigned (by an expert), representing the importance of the data in the class. A combined quantitative value could then be calculated by the probability and the accumulated importance of the processed assets. This approach is similar to risk calculation in the business domain. The developers can be guided to the highest resulting combined value.

From a project management point of view, the effort for the verification of a method should also be taken into account. Therefore, when two methods are similar or equal in the calculated importance for verification, the verification effort can then help as a further decision factor. However, in semi-automatic or manual

verification approaches, this effort depends not only on the complexity and the size of the source code, but also on the experience and skill of the verification expert. Therefore, it is difficult to quantify the effort for the verification of source code parts.

With the quantitative values of the presented analysis, the developers can be provided values to guide them to locations in the source code that should be prioritized for measures to secure the system. In case no further information about the system is available, the developer can be guided to the location with the highest calculated probability. However, taking *only* the probability of entering the critical regions of a method into account can result in a bad guidance. For example, in a banking scenario, two methods can leak information about a client when a wrong input is made. In a given usage profile, the first method leaks the address of the client to an employee with a probability of 20% and the other method leaks the complete credit card data instead of only the last 4 digits of the client with a probability of 2%. When only regarding the probabilities for the leakages, the developer would be guided to the error that leaks the address of the client. Therefore we suggest to use the risk as guiding values as before. Again, the risk is defined as the probability of occurrence together with the impact when executing the critical region. The damage that is done be executing the critical region can be defined by an expert or by tools that analyze if a certain security requirement breaks. In the example of the banking scenario, when the critical region leaking the credit card data is assigned a high value and only a small value is assigned to the leak of the address data, the developers can be pointed to the more important leak of the credit card data. This process can be further refined when considering attackers in the system as in Fig. 9. Because attackers select the greatest damage, all components reachable by an attacker should be secured first, beginning with the highest risk.

7 Conclusion

This report demonstrates the possibility of the correctness quantification on the software architecture level. We show how to combine static analysis tools for source code and analysis tools for models, to measure a degree of correctness given a predefined model of usage. The approach is applicable in all phases of software development, but with a different degree of precision – depending on the knowledge about the system usage and the critical regions of the software. Also we show how the approach can be exploited for measuring security properties. Furthermore we described how the values obtained with the approach can be used to guide developers to locations where verification can be applied and to locations that should be secured.

Our approach is currently incomplete, as many features of software system are not covered. The following list gives insights on the next required steps:

Stateful services. In the current approach, our services are considered to be

stateless limited by the critical region that only a Boolean expression over the input parameters. Lifting this limitation requires more than just adding state variables to the critical region. We also need to model the call history of the services and the call effects on the states. Additionally, we also need to consider that the services are used by multiple user simultaneously. Research is necessary to (1) include state variables in PCM, and (2) to adapt the critical region inference in KeY.

Generation of critical region. We considered only KeY for the generation of critical regions. But less complex facilities could also provide them. For example, we can include positive and also negative test cases generated by hand or fuzzing.

Probabilistic capabilities in Palladio. The stochastic expressions of PCM are limited, e. g., conditional or joint probability distribution are not expressible. These expressions should be extended, and maybe a new probabilistic reasoning machine may be required.

Multi-party Usage Models. Currently, the usage model only describes the observed behaviour in the form of service calls. This model does not include information as to who is calling the service. For a more fine-grained specification of compromised users or roles, or to specify information flow, we consider taking external service callers into account.

Lightweight modeling of critical regions We model the critical regions into the SEFFs in a heavyweight manner by using Branches which lead to doubling in size. However, for providing a leaner model, it is possible to introduce a new model element in the PCM that captures conditions for incorrect behavior and extend the calculation rules of the *DependencySolver*.

Inference of SEFFS from source code Modeling of software is a costly and time-consuming process. Therefore, modeling is sometimes neglected and only source code is produced. Furthermore, when evolving a system and modifying the source code, changes to the models have to be adapted e.g. by a consistency preservation tool. To support the given approach in a more concise manner, it could be possible to use the available source code together with the provided conditions of KeY to directly generate the required SEFF instead of adding the conditions afterwards. When introducing the model element to capture conditions for incorrect behaviour, the analysis result directly relates to this element. Also, it would be possible to use a consistency-preserving approach like the Vitruvius-Approach that keeps the models consistent.