Leveraging State to Facilitate Separation of Concerns in Reuse-oriented Performance Models

Dominik Werle
dominik.werle@kit.edu
Karlsruhe Institute of Technology

Stephan Seifermann, Sebastian D. Krach
{seifermann,krach}@fzi.de
FZI Research Center for Information Technology

Abstract

Each of the five dedicated roles of the Palladio process considers one or more concerns that form a performance prediction model, altogether. Modeling systems that vary their behavior based on a request history, however, requires to break role separation and create dependencies between concerns, thus reducing the reusability of components. Model elements that allow expressing such behavior while maintaining role separation do not exist. We propose a model extension that allows expressing behavior statefully and a transformation to a basic stateless Palladio model. This allows to maintain the role separation and thereby the reusability of components without the need for changes of existing analyses.

1 Introduction

Palladio enables software architects to predict the performance of a component-based system before implementing or deploying it. The Palladio performance analysis requires performance abstractions delivered by five dedicated roles [3, pp. 203-205]: (1) software architects, (2) component developers, (3) system deployers, (4) domain experts, and (5) quality analysts. Each role owns specific knowledge about the system and is active in different stages of the development process. The separation between the component developers and the other roles is important to facilitate component reuse by hiding implementation details behind defined interfaces.

However, we observed that predicting performance for system behaviors dependent on previous requests often weakens this separation. Heinrich et al. [2] identified the need for explicit modeling of this type of behavior as well for expressing *queue-dependent behavior*. For instance, there are two modeling approaches for systems that behave differently after k requests by a user that both violate role separation: (a) The domain expert has to model two usage scenarios (USCs): One regular and one misuse scenario. The latter includes k regular and the remaining amount of alternative requests explicitly. This carries information from the component behavior into the usage model. (b) The component developer has to integrate the amount of alternatively handled requests into a branch probability. This requires modeling user behavior in the component model, thus impeding the component reuse.

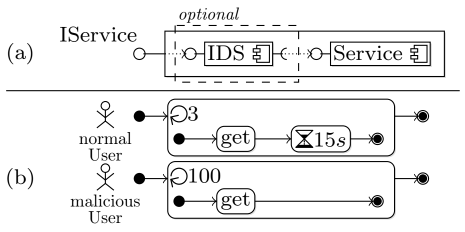


Figure 1: Example (a) system and (b) usage models

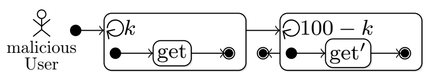


Figure 2: Usage scenario to describe IDS behavior

In this paper, we propose *Session Count Expressions (SCEs)* that enable state-based behavior descriptions using call counters. Thereby, component developers and domain experts do not have to intertwine their concerns anymore. We apply the modeling extension for a specific kind of state in an example. Additionally, we sketch a transformation from the newly introduced elements to existing ones. Therefore, analyses do not have to be altered, while the semantics of a stateful model are approximated.

The remainder of the paper is structured as follows: In Section 2, we introduce a running example for the violation of the role separation. Section 3 covers existing solutions (3.1) and our proposed solution (3.2). We conclude the paper in Section 4.

2 Exemplary Violation of Separation

In the following, we introduce a minimal example that illustrates the issue. We omit the models of the resource environments and the deployment in favor of comprehensibility. Figure 1 (a) shows a service interface `IService` with a single method `get` and a component `Service` that provides it. In this example, we want to investigate the performance impact of the addition of an intrusion detection system (IDS). In our scenario, the IDS acts as a proxy for `Service` and passes requests from the system interface to the implementing component. Our usage model of the system (Figure 1 (b)) contains two USCs: `normalUser` is a user that calls the method `get`, waits 15 seconds, and repeats this twice. `maliciousUser` is an attacker that performs 100 calls to the method in quick succession to impair the performance of the system.

We assume that our IDS operates according to the following heuristic: If the requesting user has already made more than k requests in the current session, the IDS performs a deep package inspection (DPI), which entails an additional overhead and can e.g. result in a message to an administrator. We do not model this alarm functionality and restrict our model to the DPI overhead. A *session* represents a USC user's lifetime.

The changed USC `maliciousUser` shows one possibility of expressing the IDS behavior in Figure 2. The USC `normalUser` is not altered. `get′` is a method added to `IService` that includes the DPI. Figure 3 (a) shows the behaviors of `get` and `get′`.

This approach violates the separation of concerns as follows. The *component developer* specifies the behavior of the implemented IDS in the modeled component. This component and its model should then be reusable by a third party without knowing its internals according to the Palladio definition of components [3, pp. 8-9]. However, the behavior of the IDS influences the usage model and therefore no *black-box reuse* is given. Furthermore, the software architect must alter the system interface to allow the call of an additional method `get′`. Additionally, the responsibilities of the roles in a component-based software development process [3, pp. 203-205] are not properly separated. The *domain expert*, who models the usage of the system, is now concerned with the inner workings of the IDS. Using only existing abstractions, we do not see an appropriate way to model this behavior without violating the presented separation of concerns.

A behavior model as depicted in Figure 4 (a) would be desirable, where the component developer can reference the number of previous calls to the method. The component developer can model an stochastic approximation of the IDS behavior as shown in Figure 4 (b). $\mathcal{U}(0, 99)$ is an uniform distribution between 0 and 99 and characterizes the number of previous calls to the method in USC `maliciousUser` without accounting for the order of calls. For the other USC, $\mathcal{U}(0, 2)$ is an equivalent approximation, that does however not express different behavior if $k > 3$. However, this also violates the separation, because the component behavior model now represents user behavior.

3 Proposed Solutions

In our scenario, the service effect specification (SEFF) must be able to distinguish between a USC execution that has already called the method k times and one that has not. Therefore, any approach that allows the representation and simulation of this kind of usage-scenario-related state is a viable means to solve the introduced problem.

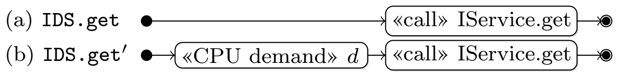


Figure 3: Behaviors of `IDS.get` and `IDS.get′`

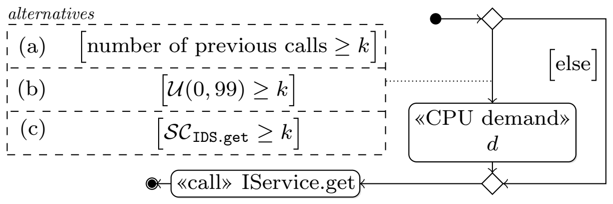


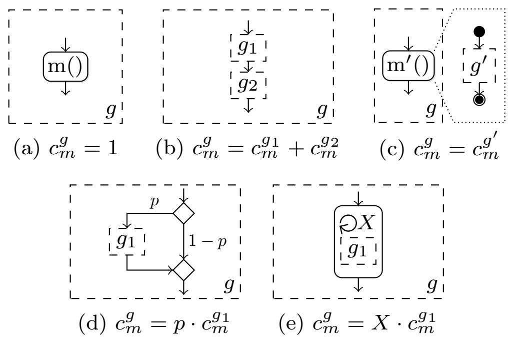
Figure 4: Alternatives for modeling `IDS.get`: (a) natural language expression, (b) uniform distribution as an approximation, (c) model with SCEs.

3.1 Existing Solutions

Stateful component-based performance models, as introduced by Happe et al. [1], allow the definition and manipulation of state related to components, systems, and users including session state. Their approach improves the result of simulations when behavior depends heavily on the state of the system. The simulator is extended to support stateful models. Analytical solving is, however, not possible anymore using the existing approaches.

3.2 Session Count Expressions

We propose *Session Count Expressions (SCEs)*, a light-weight modeling construct for expressing behavior depending on the number of requests originating from the current USC execution. A transformation resolves SCEs to stochastic expressions before the analysis of the model. Thus, existing simulations or analyses can be reused. In contrast to stateful simulation, which allows deterministic simulation of state, SCEs stochastically approximate stateful behavior. The transformation is transparent to the user and does, therefore, not violate the separation of roles. After introducing SCEs and the transformation, we show



(a) $c_m^g = 1$ (b) $c_m^g = c_m^{g_1} + c_m^{g_2}$ (c) $c_m^g = c_m^{g'}$

(d) $c_m^g = p \cdot c_m^{g_1}$ (e) $c_m^g = X \cdot c_m^{g_1}$

Figure 5: Subgraphs g and corresponding values of c_m^g

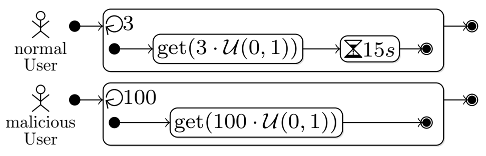


Figure 6: Stochastic approximation of the USCs

how to use the proposed construct to specify the behavior for our running example.

SCEs make an additional term $\mathcal{SC}_m$ available in stochastic expressions. These expressions are used in the specification of USCs and SEFFs. $\mathcal{SC}_m$ is a random variable that characterizes the amount of previous calls to a method m during the current USC. The characterization depends on its environment in the USC or SEFF. In the following, we use the term *subgraph* to refer to a connected subgraph of steps in a USC/SEFF. To derive $\mathcal{SC}_m$, we first describe how to determine c_g^m, which characterizes the number of calls to m in a subgraph g. This process is illustrated in Figure 5. (a) The base case of a call to m results in $c_m^g = 1$. (b) For subsequent subgraphs, the number of calls can be summarized ($c_m^g = c_m^{g_1} + c_m^{g_2}$). (c) For calls to methods m' ($m' \neq m$), we derive $c_m^g = c_m^{g'}$ from the behavior of m'. (d) A branch with probability p results in $c_m^g = p \cdot c_m^{g_1}$. We restrict our description to probabilities branch conditions that are not dependent on input parameters. (e) For a loop action with iteration count specified by the random variable X, $c_m^g = X \cdot c_m^{g_1}$. Similar to the simulative performance prediction, the derivation does not support (direct or indirect) recursion of calls to m.

A transformation that reduces a model with SCEs to a model without them works as follows. (1) For each use of a random variable $\mathcal{SC}_M$ in a SEFF, add a parameter p_M^{in} to *all* signatures of *all* methods in the model, (2) replace each reference to $\mathcal{SC}_M$ inside a SEFF by $p_M^{in} + c_M^g + \mathcal{L}$ where g is the subgraph of all preceding actions. The *loop approximation* is $\mathcal{L} = 0$, except for references to $\mathcal{SC}_M$ *inside* loops. In favor of comprehensibility, we will not explain the general case of arbitrary subgraphs that call M before or after the current reference to $\mathcal{SC}_M$ inside the loop. Instead, we will focus on the case where M is only called once inside the subgraph and all references to $\mathcal{SC}_M$ are *after* the call. Then, we can derive an approximation for the number of previous calls *inside* the loop by sampling a uniform distribution between 0 and the number of iterations X minus 1 which gives us $\mathcal{L}$. (3) For each external service call in a SEFF and for each service call in a USC, pass $p_M^{in} + c_M^g + \mathcal{L}$ as the parameter.

Using this transformation, we can for example refer to $\mathcal{SC}_{\texttt{IDS.get}}$ in our behavior description as depicted in Figure 4 (c). The parameter $p_{\texttt{IDS.get}}^{in}$ is added to the signature of the method. Because there are no other calls to $\mathcal{SC}_{\texttt{IDS.get}}$ in the model ($c_M^g = 0$) and we are not in a loop ($\mathcal{L} = 0$), we derive $\mathcal{SC}_{\texttt{IDS.get}} = p_{\texttt{IDS.get}}^{in}$. Figure 6 shows the transformed USCs. The arguments are the approximated number of calls to `IDS.get` ($\mathcal{SC}_{\texttt{IDS.get}}$) for the normal and malicious scenario respectively ($c_{\texttt{IDS.get}}^g = 0$, $\mathcal{L} = 2 \cdot \mathcal{U}(0,1)$ resp. $99 \cdot \mathcal{U}(0,1)$). Note that the transformed normal USC models the same behavior as before, because the distribution only yields values smaller than k.

4 Conclusion

In this paper, we demonstrated a shortcoming regarding black-box reusability caused by stateless behavior modeling of Palladio. More precisely, we have shown an example, where the component behavior influences the usage model and vice versa. As an alternative to stateful behavior simulation [1], we introduced SCEs, a light-weight modeling extension that can be transformed to stateless stochastic performance models without violating the separation of concerns.

Enabling the description and simulation of state in a performance model allows a cleaner separation of concerns regarding the roles involved in a component-based software development process. Our proposed extension supports this separation for behavior based on request amounts while maintaining support for existing analyses. Additionally, we demonstrated that modeling of behavior which depends on *sets* of requests, including for example data streams or flows, is not only necessary to analyze new quality properties such as security, but can also help to align system development with the component-based software development process of Palladio.

Next steps encompass generalizing the approach to other stochastic performance modeling formalisms besides Palladio, the implementation of the transformation, and the comparison of the accuracy of the transformed models with models that simulate state, as it is done in stateful Palladio [1]. Additional transformation rules for other kinds of state (component, system, user) would allow a more comprehensive use of the approach.

Acknowledgements. This work was partially funded by the German Research Foundation (DFG) as part of the Research Training Group GRK 2153: "Energy Status Data – Informatics Methods for its Collection, Analysis and Exploitation".

References

[1] L. Happe, B. Buhnova, and R. Reussner. "Stateful component-based performance models". In: *Software & Systems Modeling* 13.4 (2013), pp. 1319–1343.

[2] R. Heinrich, H. Eichelberger, and K. Schmid. "Performance Modeling in the Age of Big Data: Some Reflections on Current Limitations". In: *Proceedings of ModComp*. 2016.

[3] R. H. Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016. 408 pp.