Chapter 8
Performance Analysis Strategies for Software Variants and Versions

Thomas Thüm, André van Hoorn, Sven Apel, Johannes Bürdek, Sinem Getir, Robert Heinrich, Reiner Jung, Matthias Kowal, Malte Lochau, Ina Schaefer, and Jürgen Walter

T. Thüm (✉) · M. Kowal · I. Schaefer
Institute for Software Engineering and Automotive Informatics, TU Braunschweig, Brunswick, Germany
e-mail: t.thuem@tu-braunschweig.de; m.kowal@tu-braunschweig.de; i.schaefer@tu-braunschweig.de

A. van Hoorn
Institute of Software Technology, University of Stuttgart, Stuttgart, Germany
e-mail: van.hoorn@informatik.uni-stuttgart.de

S. Apel
Chair of Software Engineering I, Department of Informatics and Mathematics, University of Passau, Passau, Germany
e-mail: apel@uni-passau.de

J. Bürdek · M. Lochau
Technische Universität Darmstadt, Fachbereich Elektrotechnik und Informationstechnik, Fachgebiet Echtzeitsysteme, Darmstadt, Germany
e-mail: johannes.buerdek@es.tu-darmstadt.de; malte.lochau@es.tu-darmstadt.de

S. Getir
Institut für Informatik, Johann-von-Neumann-Haus, Humboldt-Universität zu Berlin, Berlin, Germany
e-mail: getir@informatik.hu-berlin.de

R. Heinrich
Institute for Program Structures and Data Organization, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
e-mail: robert.heinrich@kit.edu

R. Jung
Software Engineering Group, Department of Computer Science, Kiel University, Kiel, Germany
e-mail: reiner.jung@email.uni-kiel.de

J. Walter
Chair of Computer Science II, Universität Würzburg, Würzburg, Germany
e-mail: juergen.walter@uni-wuerzburg.de

© The Author(s) 2019

R. Reussner et al. (eds.), *Managed Software Evolution*,
https://doi.org/10.1007/978-3-030-13499-0_8

Adaptation is heavily used for today's software in two dimensions. First, developers frequently release new *versions* of software to meet new or changed requirements (aka. software evolution [BR00]). Second, developers simultaneously develop *variants* of software to meet contradictory requirements (aka. configurable software or software product lines [CE00, Ape+13]). While versions typically replace existing versions, variants co-exist to meet certain requirements each. Both variants and versions give rise to software variation. Performance—capturing software quality properties with respect to timeliness and resource usage—is of particular relevance to software design, operations, and evolution. It has a major impact on key business indicators. Consequently, during the software's life cycle, developers and operators need to be aware of performance.

Over the last decades, the community has developed methods, techniques, and tools to analyse performance in different design and operations stages, combining model-based and measurement-based approaches [WFP07, Bru+15]. Figure 8.1 depicts the artefacts and activities being involved in model-based performance analysis in combination with measurements. architectural models for the software versions and variants, for example using Unified Modeling Language (UML), can be augmented by performance-relevant information, for example using UML profiles such as MARTE [Obj11]. These models can be used to predict performance indices of the respective versions and variants, for example CPU utilisation and response times. Two common approaches are used for prediction [CMI11]: (1) simulating the models and (2) transforming the architectural models to analytical models, for example queuing networks or Petri nets and solving or simulating these models using respective tools. Once implementation artefacts become available, performance indices can be obtained by measurements, for example using profilers or application performance management (APM) tools [Heg+17]. Once measurements are available from implementation artefacts, performance models can also be

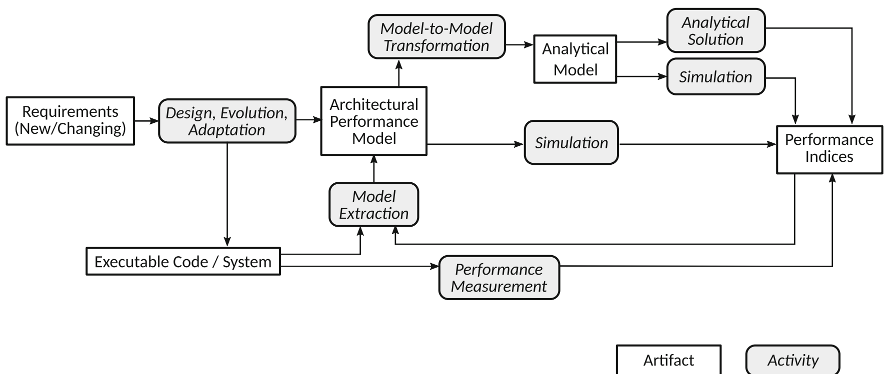

Fig. 8.1 Performance engineering taxonomy including alternative performance evaluation methods

extracted automatically. These extracted models can also be used during runtime, example to react dynamically to changing environmental conditions, such as changing workload characteristics [CMT16].

Even though variants and versions are quite different in their purpose, their software variation challenges software analyses in a similar manner. In particular, it is often infeasible to analyse all variants and all versions of a software, especially for performance analyses, because of several reasons. First, even when applied automatically, performance analyses are time-consuming due to the necessity to execute the software under test using different workloads [WFP07]. Second, the sheer number of variants and versions of today's software renders it infeasible to analyse all of them separately due to combinatorial explosion [Thü+14a]. Even though variation is often low between certain variants and versions, a small change can have a huge impact on the performance of the overall software. Hence, we cannot just measure the performance of one variant or version and, thus, need strategies to systematically cope with software variation.

Ideally, the performance of software variation would be analysed with an automated process that incorporates the knowledge of previous performance analyses steps. We envision a process in which a stakeholder identifies a performance-related concern. Then a magic box automatically selects a strategy to answer the concern, including a mixture of predictions, as well as offline and online tests. When applying this strategy, results are not only propagated to the stakeholder but also to a knowledge base. While the stakeholder acts on the results by evolving the system or refining concerns, the growing knowledge base is used by the magic box in the next iteration.

In this chapter, we report on our experiences with performance evaluation strategies for software variation. We elaborate on strategies to efficiently analyse the performance of software variants in Sect. 8.1 and of software versions in Sect. 8.2. We are using both case studies introduced in Chap. 4 for illustration. Section 8.1 is exemplified using the Pick-and-Place Unit (PPU) case study, while Sect. 8.2 uses the Common Component Modeling Example (CoCoME) case study. We conclude our discussions by giving a unified view over performance analysis strategies for software variation and a discussion of future challenges in Sect. 8.3.

8.1 Analysis Strategies for Software Variants

Numerous strategies are known to analyse software variants [Thü+14a]. However, not all of them are applicable for performance evaluation, as some strategies can be used only for static analysis and not to actually run the software variants. We report on our experience in applying complementary strategies to analyse the performance of software variants. In Sect. 8.1.1, we elaborate on approaches that try to focus on the most relevant variants by sampling the large variant space. As we use test cases to measure the performance of variants and as manually creating those is laborious, we discuss how to generate test suites that cover all variants in Sect. 8.1.2.

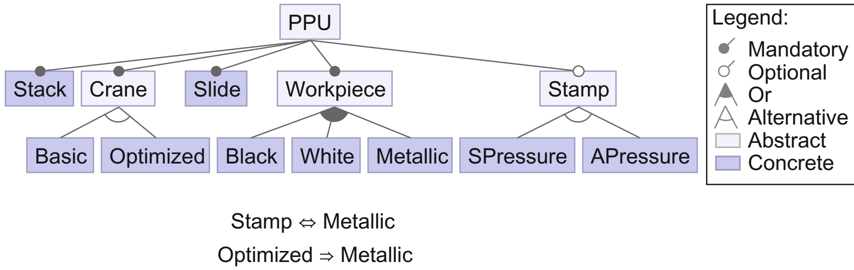

Fig. 8.2 Excerpt of the feature model of the PPU

Finally, in Sect. 8.1.3, we discuss a strategy to predict the performance of variants analytically without the need to measure the performance of every variant. That is, the techniques discussed in Sects. 8.1.1 and 8.1.2 may or may not be combined, whereas the technique presented in Sect. 8.1.3 is applied in isolation.

Pick-and-Place Unit as a Motivating Example A feature model typically has a tree-like graphical representation depicting the hierarchically arranged set of features. Relationships in the feature model regarding parent and child features are expressed with the common notation of *mandatory*, *optional* features and *or*-, *alternative* groups and their underlying semantics (cf. legend in Fig. 8.2 for the graphical representation) [Kan+90a, CE00]. *Abstract* features do not contain realisation artefacts and are only used for structural purposes [Thü+11]. A feature model of the PPU case study system, as introduced in Chap. 4, is shown in Fig. 8.2. The PPU can process up to three different kinds of workpieces (WPs): *White*, *Black*, and *Metallic* workpieces. A *Stack* stores all workpieces before they are processed by the *Crane*. *Basic* and *Crane* are two alternative implementations of the crane behaviour differing in the processing times of workpieces. In addition, the optimised implementation requires a stamping module, making the metallic workpiece type necessary. Finally, all workpieces are transported to the *Slide*, awaiting packaging or further processing in other automation systems. For illustration, we describe three selected variants in more detail in the following.

Variant 1 is the minimal system configuration consisting of the concrete features *Stack, Basic, Slide,* and *Black*. The *Black* workpieces are transported from the *Stack* to the *Slide* by the *Crane*. This process is repeated until no more workpieces are present.

Variant 5 can distinguish between two different types of workpieces (*Metallic* or *Black*). While *Black* workpieces are treated as in Variant 1, metallic pieces take a different route through the system. They are transported by the *Crane* to the new stamp component (i.e. *SPressure*). After the stamp process is finished, these pieces are also transported to the *Slide*.

Variant 9 is identical to the previous variant on the hardware level. The crane implementation is optimised (cf. feature *Optimised*) as the crane no longer waits at the stamp for the stamping process to be finished. Instead, the crane moves

back to the stack to pick up the next black workpiece (if present) and transports it to the slide. Afterwards, the crane fetches the stamped workpiece and transports it to the slide.

8.1.1 Sample-Based Analysis of Software Variants

As said previously, configurable systems may have configuration spaces of substantial size, so identifying which variant performs best in a concrete setting is difficult. In the worst case, the size of the configuration space of a configurable system is exponential in the number of features. While, in practice, the actual number of desired or relevant software variants is considerably smaller, typically, configuration spaces of real-world systems are still huge [Ber+13]. In fact, even enumerating all valid variants—not to speak of performing any measurements—is often computationally intractable. Due to the small size of our example, enumerating all variants is possible though, as we illustrate in Table 8.1.

To learn about the performance behaviour of individual variants, practitioners resort typically to *sampling*. The idea is not to analyse all variants of a given configurable system individually, but just a *sample set*, which is smaller and can be analysed in feasible time. For the purpose of our example, let us assume that we analyse variants 1, 4, and 9 (cf. Table 8.1). The key idea of a sampling-based approach is not just to work with the performance data of the sample set but to use them also to learn about the performance behaviour of other variants not in the sample set, say variants 7 and 10, in our example. In other words, we want to *predict* the performance behaviour of all (or some) variants of a system based on the performance measurements we did on a sample set.

Sampling Strategies There are various strategies to select a proper sample set and to generalise the measurements to the other variants of the system. Let us illustrate some key strategies here by means of the example of Table 8.1, which includes a performance value for every variant of the PPU case study. An interesting observation is that there are only three different kinds of variants: variants that can process 0.12 workpieces/s, 0.03 workpieces/s, and 0.09 workpieces/s. Interestingly, in our sample set (variants 1, 4, 9), there is no variant with the value 0.09. While this is not necessarily a problem, we will we discuss it shortly as it illustrates that selecting variants for the sample set is a crucial step.

In the literature, there are several strategies for selecting sample sets [Med+16]. One notable strategy—beyond mere *random sampling*—is *t-wise coverage* sampling [JHF12]. The idea is that the variants of the sample set should contain or *cover* certain features and combinations of features. *Feature-wise* ($t = 1$) sampling means essentially that every feature of the configurable system should be selected in, at least, one variant and deselected in, at least, one variant of the sample set. In our exemplary sample set, this is not the case as, for example, feature *APressure* is not in any of its variants. In contrast, sets 1, 11, and 12 are a valid feature-wise sample

Table 8.1 Variants of the pick-and-place unit and their performance values

Variant	Concrete features	Performance (in workpieces/s)
1	Stack, Basic, Slide, Black	0.12
2	Stack, Basic, Slide, White	0.12
3	Stack, Basic, Slide, Black, White	0.12
4	Stack, Basic, Slide, Metallic, SPressure	0.03
5	Stack, Basic, Slide, Black, Metallic, SPressure	0.09
6	Stack, Basic, Slide, White, Metallic, SPressure	0.03
7	Stack, Basic, Slide, Black, White, Metallic, SPressure	0.09
8	Stack, Optimised, Slide, Metallic, SPressure	0.03
9	Stack, Optimised, Slide, Black, Metallic, SPressure	0.12
10	Stack, Optimised, Slide, White, Metallic, SPressure	0.03
11	Stack, Optimised, Slide, Black, White, Metallic, SPressure	0.12
12	Stack, Basic, Slide, Metallic, APressure	0.03
13	Stack, Basic, Slide, Black, Metallic, APressure	0.09
14	Stack, Basic, Slide, White, Metallic, APressure	0.03
15	Stack, Basic, Slide, Black, White, Metallic, APressure	0.09
16	Stack, Optimised, Slide, Metallic, APressure	0.03
17	Stack, Optimised, Slide, Black, Metallic, APressure	0.12
18	Stack, Optimised, Slide, White, Metallic, APressure	0.03
19	Stack, Optimised, Slide, Black, White, Metallic, APressure	0.12

but still do not contain a variant with the value 0.09. *Pair-wise* ($t = 2$) sampling requires that for each pair of features there is at least one variant, in which both are selected and both are deselected and each feature is selected while the other is deselected. Our exemplary sample set does not attain pair-wise coverage either as it does not even cover all features. The pair-wise sample sets 1, 2, 3, 6, 9, 12, 16, and 19 would be sufficient for our example. Selecting higher values of t increases coverage but also leads to larger sample sets.

Learning from Sample Sets Given a sample set, there are several approaches that aim at learning the influences of individual features and their combinations on performance to allow predictions of the performance behaviour beyond the sample set [Sie+12a, Guo+13, Sar+15, Sie+15, Nai+17]. A simple approach is to approximate the performance of every individual feature [Sie+12a]. This can be achieved easily by a comparative measurement: measuring a basic variant with and without the feature in question and assigning the difference in performance

behaviour to that very feature. As an example, let us assume that we measured (denoted using function Π) the processing time of variants 6 and 7 of our PPU case study (in workpieces per second):

$$\Pi(\text{Stack, Basic, Slide, White, Metallic, SPressure}) = 0.03$$
$$\Pi(\text{Stack, Basic, Slide, Black, White, Metallic, SPressure}) = 0.09$$

The difference in observed throughput is 0.06, which we consider as the influence of the feature black (as it is the only feature in which the two configurations differ). This way we can assign every feature a value. Based on the values for individual features, we can already make predictions, which are rather imprecise, though. For example, if want to predict the combined influence of the features black and white, we would just add their individual influences, say $0.06 + 0.09 = 0.15$ (assuming the individual influence of white is 0.09).[1] The point is that this prediction may be wrong (in fact, it is very likely wrong). The reason is that the two features may *interact* interfering at the level of processing time (or other properties).

Feature Interactions Let us revisit the prediction procedure: Essentially, it takes the influences on the processing time of individual features and adds them up according to the variant whose processing time shall be predicted. However, due to feature interactions, the influences of the features involved do not necessarily add up, as we have seen for the features black and white. To identify the interaction between the two, we need to measure a variant that has both features selected, in addition to the measurements that we already have. This way we can pinpoint the interaction, which amounts, say, to a decrease of 0.03 workpieces/s. Knowing the influence of this interaction, we can make a more precise prediction, which is $0.06 + 0.09 - 0.03 = 0.12$.

So incorporating feature interactions improves the accuracy of the prediction procedure. The downside is, to identify all feature interactions of a configurable system, we need again to measure a possibly exponential number of system variants. This is where the sampling strategies come into play. Using, for example, pair-wise sampling presumes that the most relevant interactions are among pairs of features, which are covered by pair-wise sampling.

Experiences and Further Reading In the course of SPP 1593, we extended the tools FeatureIDE and SPL Conqueror. FeatureIDE is an Eclipse-based development environment for feature-oriented software development [Thü+14b, Mei+17], in which we integrated numerous sampling algorithms [AlH+16b, AlH+16a]. We used FeatureIDE to compute the samples for our running example. SPL Conqueror bundles various sampling and learning strategies for the performance prediction of configurable systems [Sie+12b].[2] In a number of studies, we applied it successfully to real-world configurable systems from different domains, including databases,

[1]Note that for other properties of interest, other ways of combining influence may be preferable, for example taking the minimum of two values for reliability.

[2]https://www.infosun.fim.uni-passau.de/se/projects/splconqueror/.

compilers, video encoders [Sie+12a, Sie+13, Sie+15], and scientific computing codes [Gre+14, Gre+17]. We further extended the whole approach, including the notion of feature interaction, to settings where numeric parameters are used to configure the system (e.g. cache size) [Sie+15], which may also interact in various ways [SSA17]. As for the learning procedure, we support classification and regression trees, linear regression, random forests, and others. As for sampling, we experimented with various coverage criteria [Med+16], as well as progressive and projective sampling [Sar+15]. Recently, we also surveyed the extensive literature on product sampling based on feature models [Var+18]. Our literature overview can be used by practitioners and researchers to find suitable sampling algorithms based on the available input, such as feature model and source code, and desired coverage criteria, such as feature interaction coverage or code coverage.

8.1.2 Family-Based Test-Suite Generation for Software Variants

The idea of sample-based performance prediction, as described in the previous Section, is to estimate performance values of all possible variants of a configurable software system, by only investigating a subset (sample) of variants. This approach enables a reduction of the overall effort required for performance analysis, as compared to explicitly considering every possible variant one by one. However, the accuracy of the predicted data naturally depends on the quality of the performance measurement data available for the sample set. Hence, experimental executions of the sampled variants are required in order to gather realistic and reliable performance measures for embedded software systems such as the PPU. To this end, the collected measurement data should rely on a high diversity of possible system behaviours, covering a high fraction of default, exceptional, and even fail-safe execution scenarios. Model-based coverage-driven testing constitutes a well-suited approach to systematically exercise the behaviours of software systems in an automated manner [UL07].

Model-Based Testing The term (software) testing in its most general form refers to any activity being concerned with investigating (and assuring) quality aspects of a given software system [UL07]. In particular, *dynamic* testing involves experimental executions of *test cases* by executing the software under controlled conditions, in order to investigate the output behaviours for particular input stimuli. The observed behaviours may comprise *functional* aspects (e.g. comparing the observed outputs to the ones expected for the inputs), as well as *non-functional* aspects (e.g. the amount of response time required by the system to produce the outputs).

Concerning model-based testing in particular, a behavioural specification (*test model*) of the software is used to automatically derive a set of test cases into a *test suite*. Test cases are usually selected into a test suite with respect to a given coverage criterion, defining a set of test goals, each to be satisfied by at least one test case of a test suite.

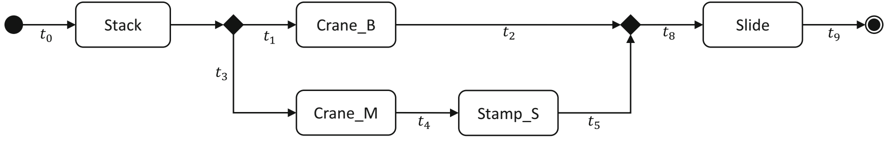

Fig. 8.3 Test model of variant 5 of the pick-and-place unit

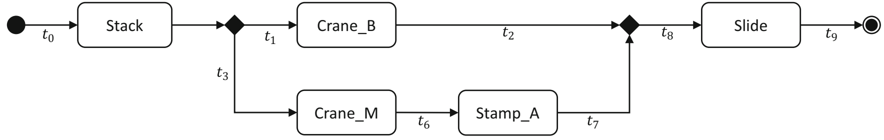

Fig. 8.4 Test model of variant 13 of the pick-and-place unit

Figure 8.3 shows an excerpt of a simplified test model for Variant 5 of the PPU, given as a UML activity diagram. However, the technique described in the following is not limited to a particular behavioural modelling language but is likewise applicable, for instance, to UML state machines and similar formalisms. The model describes the scenarios for the treatment of *Black* workpieces (branch with action *Crane_B*), as well as *Metallic* workpieces (branch with action *Crane_M*, followed by *Stamp_S*) both coming from the *Stack* and subsequently going to the *Slide*. A test case, therefore, consists of a sequence of actions from the initial action to the final action of the activity, connected via a path of control-flow edges. For brevity, we omit further details about the actions performed and the edge labels in the following examples (cf. [Loc+14] for further details). As coverage criterion, we consider edge coverage, where the set of test goals is annotated as $t_0, t_1, \ldots, t_9$ in Fig. 8.3. A test case derived from the test model for reaching, for instance, test goals t_9 may be given as the sequence $T1 = (t_0, t_1, t_2, t_8, t_9)$. This test case also covers test goals t_0, t_1, t_2 and t_8, whereas test goals t_3, t_4 and t_5 of the alternative branch remain uncovered. Hence, in order to also cover the alternative branch, a further test case $T2 = (t_0, t_3, t_4, t_5, t_8, t_9)$ is required such that a test suite consisting of $T1$ and $T2$ achieves complete edge coverage.

Considering Variant 13 of the PPU (cf. test model variant in Fig. 8.4), the behaviour corresponding to test case $T1$ remains the same (and may, therefore, be reused for also testing this variant). In contrast, the behaviour of $T2$ is not valid any more as metallic workpieces are now treated differently by the *Stamp*, thus requiring an additional test case $T3 = (t_0, t_3, t_6, t_7, t_8, t_9)$. Nevertheless, re-generating a test suite anew from scratch for every individual variant in order to finally achieve complete coverage on all variants tends to become inefficient [Bür+15a]. This is due to the high amount of similarity among the variants leading to a potentially high number of redundant test cases. In addition, in case of configurable software of realistic sizes, this approach even becomes impossible as the number of variants potentially grows exponentially in the number of features.

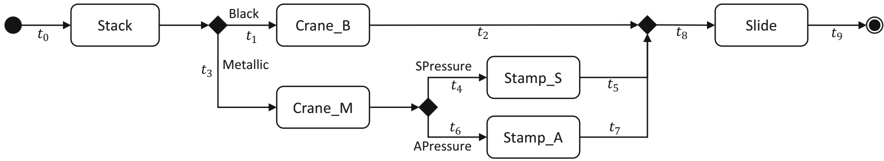

Fig. 8.5 150% test model of the pick-and-place unit

Table 8.2 Test models for variants of the pick-and-place unit

	Features				Edges									
Variant	Black	Metallic	SPressure	APressure	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
1	x				x	x	x						x	x
4		x	x		x			x	x	x			x	x
5	x	x	x		x	x	x	x	x	x			x	x
12		x		x	x			x			x	x	x	x
13	x	x		x	x	x	x	x			x	x	x	x

Family-Based Test-Suite Generation Family-based product-line analysis in general [Thü+14a] and family-based test-suite generation in particular [Bür+15a] aim to automatically achieve complete test coverage for all variants without considering every variant individually. To this end, a so-called 150% test model is used that superimposes all test-model variants into one integrated test model. An excerpt from the 150% test model of the PPU example is shown in Fig. 8.5, comprising behaviours for variants 1, 4, 5, 12, and 13. Variable parts (e.g. edges in this example) of a 150% model are augmented with *presence conditions* (i.e. propositional formulae over Boolean feature variables), denoting those subsets of configurations, in whose corresponding test-model variants the respective part is present.

Table 8.2 summarises for the set of all variants which edges (and, therefore, which test goals) are present (or relevant) in which variant. This additional information can be utilised during test-case generation for reasoning about the reuse of test cases while covering test goals in different variants. For instance, test goal t_1 is only present in variants with feature *Black* being selected (i.e. variants 1, 5, and 13), whereas t_0 is present in all variants. The aforementioned test cases $T1$ (requiring feature *Black*) and $T2$ (requiring features *Metallic* and *SPressure*), therefore, together cover test goal t_0 on variants 4 and 5, but they are both not valid for variants 12 and 13 (requiring feature *APressure* to be selected). Hence, a third test case, $T3 = (t_0, t_3, t_6, t_7, t_8, t_9)$, is to be derived to finally cover test goal t_0 on all variants in which it occurs.

The possible reuse of test cases among variants sharing similar paths achievable by family-based test-suite generation potentially reduces testing effort as compared to variant-by-variant testing. For instance, applying variant-by-variant test-case derivation to the PPU example in Table 8.2 at least produces an overall number of seven test cases (i.e. $1(V1) + 1(V4) + 1(V5) + 2(V12) + 2(V13) = 7$) according to the number of paths within the different test-model variants. In contrast, when

Table 8.3 Test suite for complete transition coverage of the pick-and-place unit test model

TC	Path	Features				Variants				
		Black	Metallic	SPressure	APressure	V1	V4	V5	V12	V13
T1	t_0, t_1, t_2, t_8, t_9	x				x		x		x
T2	$t_0, t_3, t_4, t_5, t_8, t_9$		x	x			x	x		
T3	$t_0, t_3, t_6, t_7, t_8, t_9$		x		x				x	x

applying the family-based test-generation strategy, three test cases are sufficient to achieve complete edge coverage on all variants, as illustrated in Table 8.3. Based on this information, two variants are sufficient to execute the resulting three test cases (e.g. variants 5 and 12).

Experiences and Further Reading Besides the PPU case study, the presented technique has been applied to other application domains, including medical-device control software, Linux-kernel drivers, and embedded-system utility software. For those experiments, we observed similar results concerning efficiency improvements as compared to variant-by-variant testing. Corresponding tool support utilises the temporal model checker SPIN for model-based (black-box) generation from UML state charts [Loc+14], as well as the software model checker CPACHECKER for white-box text generation from product lines implemented in C using compile-time variability (C pre-processor) [Bür+15a]. Our experience gained from the various experimental results show that remarkable efficiency improvements of family-based coverage-driven test generation, as compared to a variant-by-variant approach, can be observed in almost all cases, at least up to a certain product-line size (concerning, e.g., the number of features and amount of code). Beyond this critical threshold, the additional effort required, for example for presence-condition analysis, may obstruct the applicability of family-based analyses. Finding a good trade-off between reuse of analysis information and scalability of family-based product-line analysis strategies therefore is the most emerging issue for future research.

8.1.3 Family-Based Analysis of Software Variants

While Sects. 8.1.1 and 8.1.2 focused on how to measure and predict the performance of variants, an orthogonal way is to build and analyse a performance model. Performance models are well understood for single systems, but applying them to each variant separately involves redundant effort. Similar to the test-suite generation of Sect. 8.1.2, we apply a family-based strategy to analyse performance models of software variants efficiently.

We extend the UML activity diagrams that are already used as test models in Sect. 8.1.2 by quantitative performance information. For instance, Fig. 8.6 depicts Variant 5 of the PPU enriched with such performance annotations. In particular, we assume that the following parameters are provided: (1) rate of arrivals of workpieces

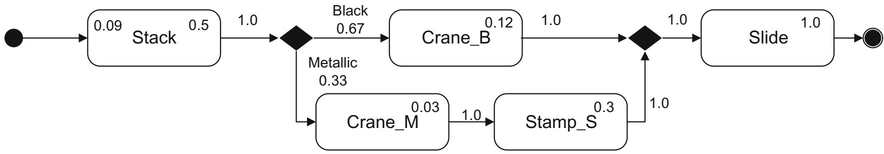

Fig. 8.6 Variant 5 of the pick-and-place unit with performance annotations

into the system, denoted by λ; hence, $1/\lambda$ is taken to be the average time between two successive arrivals at the system (cf. Fig. 8.6, top-left corner of the initial node), and (2) rate of processing a workpiece by each node, denoted by μ (cf. Fig. 8.6, top-right corner of a node). Finally, we require annotations on the edge connecting nodes. Specifically, an edge between nodes V_i and V_j must be annotated with the probability that a workpiece processed by node V_i goes to V_j. We call this model a performance-annotated activity diagram (PAAD). Once the annotations are made, the PAAD is amenable for an automatic performance evaluation. In particular, we can interpret a PAAD as a continuous-time Markov chain with an underlying Jackson-type queuing network [Jac63]. The evaluation is executed by solving the following system of equations: $(I - P^T)\gamma = \lambda$. P is the routing probability matrix, I is the identity matrix, λ is a vector based on the defined arrival rates, and γ is a vector containing the effective arrival rates that we are interested in. For instance, the considered PPU variant gives us

$$P = \begin{bmatrix} 0.0 & 0.67 & 0.33 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \qquad \lambda = \begin{bmatrix} 0.09 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} \qquad \mu = \begin{bmatrix} 0.5 \\ 0.12 \\ 0.03 \\ 0.3 \\ 1.0 \end{bmatrix}$$

Once we solve the system for γ, the steady-state behaviour of the network is fully characterised and we can interpret the results of the analysis in terms of user-perceivable performance properties of the system [Ste09]:

- *Throughput:* the number of workpieces that a node can process in a given amount of time (i.e. γ)
- *Utilisation:* the probability that a node is busy processing a workpiece (i.e. γ/μ whereas μ is the service rate)
- *Queue length:* the number of jobs waiting at a node, including those in service (i.e. $\gamma/\mu/(1 - \gamma/\mu)$).

For instance, the utilisation of each node in Fig. 8.6 is computed with 18% for the *Stack*, 50% for the *Crane_B*, 100% for the *Crane_M*, 10% for the *Stamp*, and 9% for the *Slide*. However, we have to solve the system of equations for each variant separately since it is not possible to reuse the numerical computations across variants. Even varying the exogenous arrival rate λ by 0.01 forces us to do a re-computation.

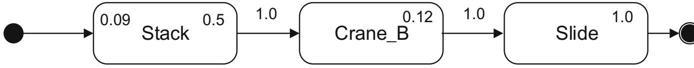

Fig. 8.7 Variant 1 of the pick-and-place unit with performance annotations

In SPP 1593, we developed a family-based performance analysis that solves the system of equations once and enables us to reuse the results across all variants. The analysis requires the construction of a 150% model of the system. Thus, we need a variability modelling mechanism in order to incorporate the individual variants into a 150% model. For this purpose, we introduced the concept of delta modelling in our approach. Delta modelling is a modular yet flexible variability modelling method on the implementation artefact level and allows capturing closed and open variant spaces. Each delta contains a set of basic operations to be performed on a PAAD, such as the addition and the removal of nodes and edges, or the modification of parameters, such as the probability of an edge and service rates in nodes. In addition, we have a core that can be an arbitrary variant of the system. Hence, applying a delta to the core yields a new variant of the system and in our case a new PAAD, which has performance characteristics that can again be numerically analysed using the product-based evaluation.

The PAAD in Fig. 8.6 represents the core of the PPU. Next, we can define a delta comprised of several transformations, that is removal of the nodes *Crane_M* and *Stamp_S* and their connecting transitions, as well as setting the probability from *Stack* to *Crane_B* to 1.0. An application of this delta to the core gives us the most basic variant of the PPU, depicted in Fig. 8.7.

We are able to model all variants of the PPU using delta modelling. Merging all deltas and the core gives us the 150% model of the system (cf. Fig. 8.5, where performance annotations are omitted). Similar to the analysis of a single variant, we can derive the routing probability matrix and vectors for arrival and service rates from the 150% model. However, each value that is different in multiple variants (i.e., depends on a delta) is now represented by a variable (i.e., symbolically rather than by a concrete value). For instance, let us consider three variants of the PPU comprised of our core (cf. Fig. 8.6), the most basic variant (cf. Fig. 8.7), and the variant introducing the second stamping module *Stamp_A* leading to the 150% model, as depicted in Fig. 8.5. The respective matrix and vectors containing symbols for each changed value are as follows:

$$P_s = \begin{bmatrix} 0.0 & p_{C_B} & p_{C_M} & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & p_{S_S} & 0.0 & p_{S_A} \\ 0.0 & 0.0 & 0.0 & 0.0 & p_{Sl} & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & p_{Sl_2} & 0.0 \end{bmatrix} \qquad \lambda_s = \begin{bmatrix} \lambda_{Stack} \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} \qquad \mu_s = \begin{bmatrix} 0.5 \\ 0.12 \\ \mu_{C_M} \\ \mu_{S_S} \\ 1.0 \\ \mu_{S_A} \end{bmatrix}$$

We solve the system of equations, but we are not able to receive concrete performance properties in terms of throughput or utilisation for individual variants at this point since the equations are solved symbolically and still contain the unknown variables from the routing matrix and the rate vectors (e.g. for utilisation):

$$Util = \left[2 * \lambda_{Stack}, \frac{25 * \lambda_{Stack} * p_{C_B}}{3}, \frac{\lambda_{Stack} * p_{C_M}}{\mu_{C_M}}, \ldots, \frac{\lambda_{Stack} * p_{C_M} * p_{S_A}}{\mu_{S_A}} \right]$$

As a final step, we have to insert the probabilities and rates of a specific variant into the symbolic solution yielding the desired concrete performance value and thus the same result as analysing each variant separately. The family-based analysis is significantly more efficient considering computation times for a given large variant space. Numerical experiments show that it can be up to two orders of magnitude faster [KST14]. The computational benefit results from the expensive process of solving the system of equations over and over again for each variant in isolation, which is not necessary in our proposed family-based analysis. In addition, the computation time giving us the symbolic solution is independent of the number of variants that are analysed afterwards. Each symbol may stand for an infinite number of values resulting in an infinite number of variants that can be analysed with the symbolic solution.

Assuming, for instance, that we wish to study the impact of different arrival rates λ into the PPU, Fig. 8.8 (left part) shows the utilisation at every node for Variant 0. The results indicate that *Crane_B* is the bottleneck of the system because its utilisation is consistently the highest. Figure 8.8 (right part) shows a similar analysis for the core (i.e. Variant 3). In this case, the bottleneck is the *Crane_M* node transporting metallic workpieces to the stamping module, which takes significantly longer compared to processing a black workpiece directly to the slide. Another scenario would be to study the distribution of black and metallic workpieces processed by the PPU in order to identify an optimal system solution. In the standard configuration, the PPU processes 2/3 black and 1/3 metallic workpieces. We can

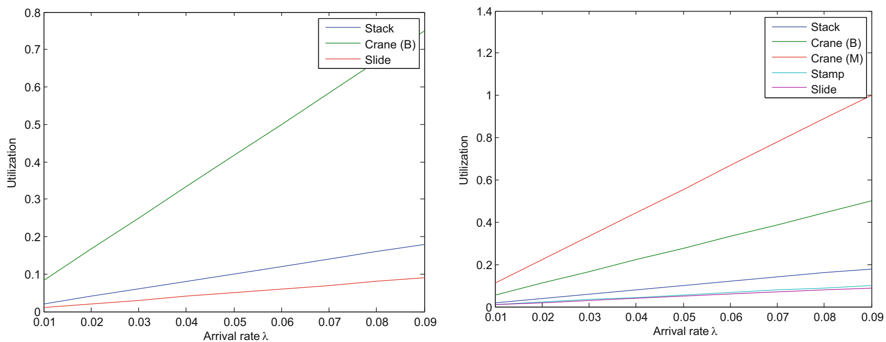

Fig. 8.8 Utilisation of variants 0 and 3 with a varying arrival rate λ

change this by simply varying the routing probabilities leading from the stack to the different cranes and look at the performance impact afterwards. We just have to plug in the desired values into our symbolic solution.

Experiences and Further Reading While we illustrated family-based prediction with the PPU, we also experimented with larger product lines with up to 430 features [KST14, Kow+15]. Especially for larger product lines, the family-based strategy significantly outperforms the separate analysis of every variant [KST14]. The above-mentioned approach has two major limitation, which we addressed by follow-up work [Kow+15]. First, service times are assumed to follow exponential distribution. Second, all computations are assumed to be performed without parallelism. For coxian-distributed multi-server stations, we measured similar performance gains of the family-based strategy [Kow+15].

8.2 Analysis Strategies for Software Versions

Section 8.1 covered performance analysis strategies for software variants, focusing on the problem of how to efficiently analyse large configuration spaces. Orthogonally, throughout the development and operation stages of a software system's life cycle, numerous software versions are created and evolved over time. With these versions, also the corresponding software artefacts (and their types) change and evolve based on the respective life cycle stages. Example types of artefacts are requirements and architectural models in the design stage, code artefacts that are available from the implementation stage, and descriptive models obtained from measurement data in the operations and maintenance stages. Connected with the changing versions and their related artefacts is the need for continuous quality assurance, for example with respect to performance as it is in the scope of this chapter.

This section covers three complementary approaches for supporting performance analyses of versions incorporating different types of software artefacts (models and code), analysis techniques (measurement-based and model-based), and suitability for the respective development stage and use case (e.g. online or offline evaluation). In Sect. 8.2.1, we present a declarative approach targeted to enable non-performance experts to select, configure, and execute performance evaluation with changing and evolving versions throughout the software life cycle. The approach presented in Sect. 8.2.2 helps to align the evolution and runtime adaptation of software versions employing models and measurements. Section 8.2.3 focuses on the co-evolution of architectural and analytical performance models.

CoCoME as a Motivating Example Figure 8.9 illustrates subparts of a performance model for the CoCoME case study, as introduced in Chap. 4. Relating to Fig. 8.1, it depicts concepts commonly found in architectural performance models, such as Palladio [Reu+16] or DML [Hub+17]. These formalisms follow common concepts known from architecture description languages (ADLs), such as config-

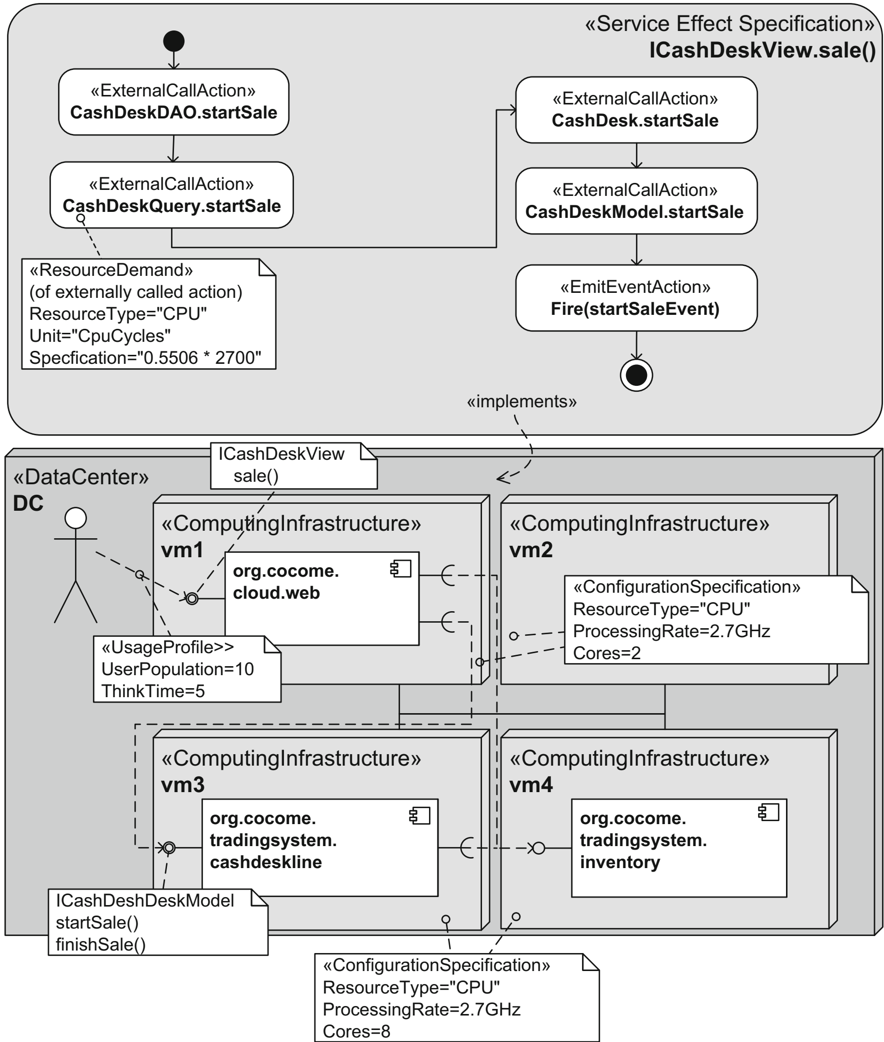

Fig. 8.9 Subparts of the CoCoME performance model in a UML-like notation

urations of components, interfaces, and connectors—presented in different views (e.g. component/connector and deployment). The example shows three CoCoME components being deployed to a networked computing infrastructure comprised of four (virtual) machines. The computing infrastructure is annotated by performance properties, such as information about the CPUs. The behaviour of the components' operations is modelled using a formalism similar to activity diagrams, including two types of actions: demands to local resources and calls to other operations. While architectural models provide a representation very close to software design models,

analytical models use abstract concepts such as resources and jobs. Their use is not limited to analysing computer systems. The models can be simulated or solved as described to predict performance indices, for example statistics about method response times, system throughput, or resource utilisation. For illustration purposes, Fig. 8.9 depicts only a subset of the complete Palladio performance model provided by the CoCoME case study.

8.2.1 Declarative Analysis Strategies for Evolving Software

During the life cycle of a software system, performance analysts repetitively need to investigate software versions to provide answers to and act on performance-relevant concerns about response times, resource utilisation, bottlenecks, trends, anomalies, etc. Their everyday work includes concerns such as *What is the response time of the CoCoME sale service? Does the CoCoME sale service satisfy its service level agreements (SLAs)? What would be the required resources to ensure the desired quality of service for the CoCoME sale service?* During the software life cycle, the evaluation of performance concerns for software versions can be based on different evaluation methodologies requiring specific performance evaluation artefacts. Supplementing measurement-based analysis, model-based predictions allow to investigate deployments, architectures, and configurations without the need to test them in a production system. Model-based performance evaluation requires a performance model. Measurement-based performance evaluation relies on a measurable system. To investigate software versions efficiently when needed requires to switch between various measurement and model-based performance evaluation approaches. Hereby, two main challenges for a continuous performance management arise:

1. *Application of performance evaluation strategies:* Holistic performance engineering applies manifold performance evaluation strategies. Each strategy is connected to particular parametrisation options and challenges, which makes them employable only with extensive knowledge and experience [Wal+16a].
2. *Selection of performance evaluation techniques:* The situation-aware choice of a performance evaluation approach is challenging. It has to consider aspects like user concerns and system characteristics to asses applicability and analysis costs.

At system design, predicting the response time of CoCoME's sale service involves complex decisions such as the selection of a suitable modelling formalism, the choice of modelling granularity, solvers and solution techniques (e.g. Markovian analytical solvers, product-form solution, or simulation-based solvers), and the derivation of model parameters.

At the system testing and deployment stages, there is the opportunity to evaluate the sales service's response time by conducting performance measurements. However, complex decisions about the measurement configuration have to be made. Decision include sufficient experiment run length, the configuration of ramp-up

time, and the choice of an appropriate instrumentation granularity allowing to obtain the required measurement data.

During system operations, it is about predicting the effects of possible system reconfigurations or the impact of an increased or changing workload mix. This enables proactive resource management but requires modelling techniques that support predicting future system states. At this time, the analysis approach and parametrisation have to be tuned for a fast response.

The concerns remain the same throughout the stages for the evolving software versions and artefacts. However, evaluation methodologies change. Selection and application affect the accuracy, as well as the speed and overhead of the analysis, and require a lot of expert knowledge.

Declarative Performance Engineering Analysing the performance of versions during the software life cycle is connected to significant efforts and complexity. Declarative performance engineering aims to provide a simplified and unified interface to investigate performance concerns for software versions abstracting from the underlying artefact and performance evaluation strategy [Wal+16a]. The idea is to use a declarative language allowing to specify performance concerns independent of the various approaches that can be applied in the context of the considered system to obtain the required information. The processing of a performance concern can be automated and optimised while hiding complexity from the user. The objective is to support system developers and administrators in performance-relevant decision-making. The declarative approach aims to reduce the huge abstraction gap between the level on which performance-relevant concerns are formulated and the level on which performance engineering techniques are typically applied. It decouples the specification of user concerns from their automated deduction. Performance concerns can be defined independent of the development stage, respective type of artefact, and evaluation method. Subsequently, suitable performance evaluation methods and techniques can be automatically selected and executed to answer the concern [Wal+18].

Expressing Performance Concerns Each version can be investigated based on different performance concerns. Figure 8.10 shows example performance concerns for CoCoME expressed using a declarative performance engineering language. Figure 8.10a shows querying of a performance metric. The processing has been constraint as `fast`, which can be interpreted by the framework to select a

```
1
2
3 SELECT sale.respTime
4 FOR SERVICE
5 "processSale" AS sale
6 CONSTRAINED AS fast
7 USING dml@'cocome';
```

(a)

```
1 EVALUATE AGREEMENTS
2 sla CONTAINS slo1
3 GOALS
4 slo1:processSale.respTime<1.3 ms
5 VARYING 'arrival rate workload'
6 AS rps <700 .. 3000 BY 500>
7 USING dml@'cocome';
```

(b)

```
1
2 MIN 'processing units cpu'
3 SATISFYING AGREEMENTS
4 sla CONTAINS slo1
5 GOALS
6 slo1:processSale.respTime<1.3 ms
7 USING dml@'cocome';
```

(c)

Fig. 8.10 Exemplary formulation of performance concerns using the declarative language. (**a**) Metric and constraint. (**b**) Contract evaluation and arrival variation. (**c**) System optimisation

fast solution strategy and configure the evaluation methodology accordingly, for example by a low required precision and a low maximum experiment run length. Figure 8.10b is about the evaluation of conformance to an SLA for different arrival rates. The concern in Fig. 8.10c proposes a resource efficient configuration that ensures conformance to SLAs. The declarative language supports further sophisticated performance analyses. It covers a wide range of performance concerns from the analysis of performance indices, aggregation, language-based system variation, determination of upper and lower bounds, SLA evaluation, threshold generation, system optimisation based on SLAs, etc.

Processing of Performance Concerns The processing of a performance concern means to automatically derive its answer. The answering process of a performance concern for a software version depends on available evaluation artefacts and situational requirements. The proposed language processing exploits a high degree of automation through a corresponding interpretation and execution infrastructure, which builds on established low-level performance evaluation methods, techniques, and tools. The architecture presented in Fig. 8.11 enables automated processing. The `Language & Editor` component provides the interface to users. The `Concern Execution Engine` provides the main execution logic. Here, all tasks independent of a specific performance evaluation technique take place. Implementations of the `Connector` interface provide functionality that is dependent on a specific performance evaluation technique. To integrate different performance evaluation approaches into the framework, multiple connectors can be subscribed at the central registry. A lean connector interface, limited to provided metrics, degrees of freedom [GBK14], and adaptations [Hub+14], allows for an easy technical connection of performance evaluation tooling to the declarative language processing framework. We provide exemplary connectors to measurement-based [Blo+16] and model-based [GBK14] analysis tooling. Besides specification of derivable indices, adaptation operations and degrees of freedom can be defined enabling additional kinds of analyses. Several analyses build upon basic performance indices. Such analyses depict reusable software parts that should be located within the `Concern`

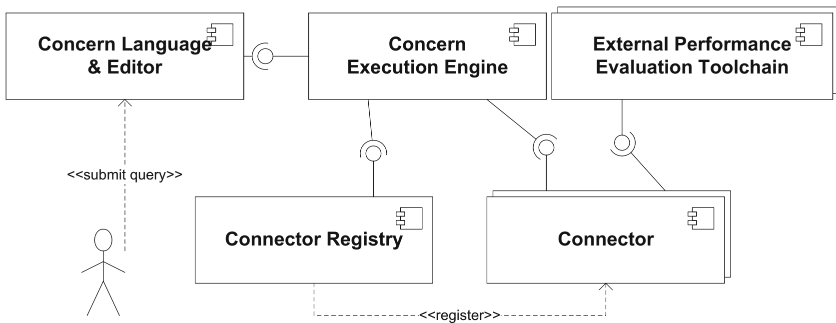

Fig. 8.11 Architecture of the DPE framework

`Execution Engine`. Indices can be forwarded to reusable algorithms, like the evaluation of SLAs [WOK17], sensitivity analysis, system optimisation [Rag+17a], etc. Also, the visualisation of analysis results can be reused independently of how values have been derived [Wal+16b].

We do not specify how to derive performance models. However, performance models are cumbersome to create manually. Therefore, automated model extraction from APM data [Heg+17], as discussed in [Wal+17a, Wal+17b], is essential to enable interchangeability of measurement and model-based performance evaluation.

Selection of Solution Strategies A solution strategy automates the answering of performance concerns by wrapping a performance evaluation method with bridging code, result filtering, and model-to-model transformations. For example, as depicted in Fig. 8.1, architectural performance models can be solved using simulation and analytical models. Existing performance engineering solution strategies come with different strengths and limitations concerning, for example, accuracy, time to result, or system overhead. While evaluation approaches allow for interchangeability, the choice of an appropriate approach and tooling to solve a given performance concern commonly relies on expert knowledge. Hence, it is a challenge to select a suitable solution strategy.

The declarative performance engineering approach allows for the automated selection of software performance engineering (SPE) approaches tailored to user concerns [Wal+16a]. To propose a solution strategy, the decision engine receives a performance concern and a description of the analysed system as input (which can be extracted from the concern definition). We provide a generic decision engine where solution strategy capability models can be registered. Instances of the capability meta-model represent analysis approaches like measurement, simulation, or analytical solvers. Compared to static decision trees, the separation of the decision engine logic and capability models allows to easily modify the description of characteristics on the evolution of performance evaluation strategies. It also facilitates the appending of additional solution strategies and rating criteria. To define capability models, we model three major aspects:

Functional Capabilities Performance evaluation approaches investigate different elements (e.g. services, processors, hard drives), metrics (e.g. response time, utilisation), and statistics (e.g. mean, sample, maximum, quantiles). The evaluable element specification integrates the named aspects and thereby defines functional capabilities of a solution approach.

Limitations The applicability of solution strategies can be limited by several constraints. Exemplary constraints for model-based analysis are on applicable input models (e.g. for product form solutions) [Bol+06] or limitations of model transformations [Bro+15]. While concepts can be transferred, measurement tools are limited to certain supported languages and technologies.

Costs Solution strategies differ in several cost types. The relevance of cost types depends on the specific application scenario. While for model-based analysis time to result is the dominating cost type, for measurement-based approaches

Table 8.4 Excerpt of capabilities for model-based analysis strategies

Analysis	Statistics	Time-to-result	Limitations
SimuCOM	Sample	High	–
LQNS	Mean	Very low	No loops, no fork-join, no parametric dependencies, no blocking-behavior
SimQPN	Sample	Medium	No loops, no fork-join, no parametric dependencies
SimQPN MVA	mean	high	No loops, no fork-join, no parametric dependencies

license costs or system overhead are the more common. Costs can either be static (e.g. fixed license costs) or dependent on the system characteristics and analysis configuration. The latter can be specified by arithmetic expressions capturing expert knowledge or various estimation techniques, for example using neural networks, machine learning, or regression approaches.

To illustrate, Table 8.4 depicts an excerpt of capabilities for analysis strategies of architectural performance models presented in [WHK17]. Supported solution strategies for the Palladio component model include SimuCOM, LQNS, SimQPN, and SimQPN MVA. SimuCOM transforms a Palladio instance to a process-based discrete-event simulation. A transformation to layered queueing networks allows triggering the analytical LQNS solver. A transformation to queueing Petri nets enables a simulation and a mean value analysis (MVA) using the SimQPN tool. Additional solution strategies for Palladio model instances, such as using SimuLizar and EventSim, can be included accordingly.

Summary During the software life cycle, multiple versions (also hypothetical ones not implemented) can be investigated for manifold performance concerns based on different evaluation artefacts. Declarative performance engineering simplifies respective analyses by automating the choice and execution of performance evaluation approaches based on a declarative specification of concerns. Other researchers have adopted the idea of declarative performance engineering also to load testing [FP18], which has not been the scope of this chapter.

8.2.2 Align Development-Level Evolution and Operation-Level Adaptation

Cloud-based software systems are subject to a wide range of changes during the operations stage [Hei16]. The usage intensity and the user behaviour that a system has to handle may change over time, which affects the system's performance. The deployment of (parts of) the software system may change, for example to address performance issues by migration and replication of components, which, however, may cause violations to privacy constraints. Execution contexts, for example virtual

machines and containers, may become available (allocation) or disappear (de-allocation), which increases or decreases the design space for system adaptation. Consequently, operating a Cloud-based software system requires to continuously observe the system and to plan for adaptation to react on changes during the operation stage that mostly cannot be foreseen during development.

This section describes how to align development-level evolution and operation-level adaptation for analysis and adaptation planning in Cloud-based software systems. In extension to Sect. 8.2.1, this section is concerned with keeping architecture and performance models for software version up-to-date while facing repeated adaptations during operations stage. Operators and developers profit from utilizing the same kind of performance models as they can communicate and exchange of knowledge based on the same abstraction. This is especially useful in fast-changing Cloud-based software systems that rely on performance analysis for their adaptation planning.

In the following, the sale service of CoCoME and the platform migration scenario are applied as a demonstrative example. In the platform migration scenario, increased usage intensity of the sale service causes an upcoming performance bottleneck due to limited capacities in the given service offering of the Cloud provider currently hosting the database. A simplified overview of the platform migration scenario is given in Fig. 8.12. For the sake of simplicity, we assume that each Cloud provider owns exactly one data centre. Different data centres are available for deploying the database service of CoCoME. We discuss how the performance bottleneck can be identified by observing the running system and be solved by planning for adaptation.

Development-Level Evolution vs. Operation-Level Adaptation Development-level evolution and operation-level adaptation can be considered as two mutual, interwoven processes that influence each other. Figure 8.13 illustrates how both processes are interconnected for Cloud-based software systems.

In addition to the several versions of the software system created throughout the development and operations stage, as introduced in the previous section, variants play a central role in Cloud-based software systems when planning for adaptations. Adaptations rely on the evaluation of alternative variants of the software's deployment and configuration to identify a new version that allows to sustain the required quality properties.

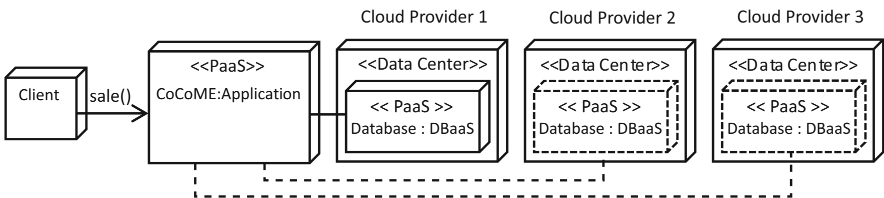

Fig. 8.12 Actual (solid line) and conceivable (dashed line) deployment in the platform migration scenario

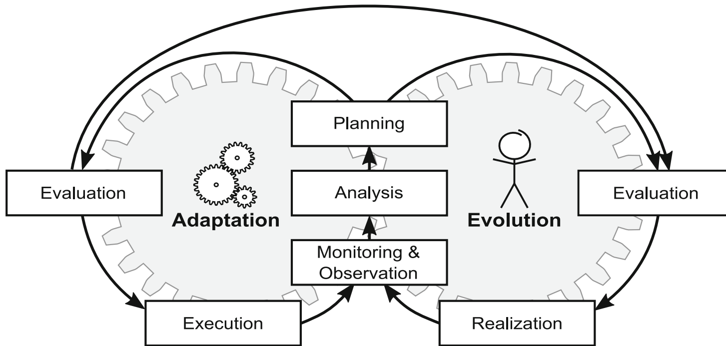

Fig. 8.13 Overview of development-level evolution and operation-level adaptation as mutual interwoven processes

Models are useful to reflect the software system and conduct analysis to identify quality flaws. A performance model of the sales services in a UML-like notation is depicted in parts in Fig. 8.9. During operations, software systems often drift away from their development models. In contrast, runtime models are kept in sync with the underlying system. Typical runtime models are close to the implementation level of abstraction. They are constructed based on observations related to source-code artefacts (e.g. service calls or class signatures) [Ben+14]. For example, observing the sales service of CoCoME results in monitoring records for the service itself and all invoked internal services. In addition, the class signature is monitored and recorded per service. While monitoring the software system, no information about its architecture is provided. Thus, it is hard to reproduce development component models from monitoring data as knowledge about the initial component structure and component boundaries is missing. This knowledge is important for system comprehension and reverse engineering. Consequently, we argue for runtime models that reflect extensive knowledge on the underlying architecture, its variability, deployment, and interaction with external services.

The iObserve Approach The iObserve approach [Has+13, Hei+14, Hei+17b] developed during the SPP 1593 addresses the aforementioned challenges by following the established MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) control loop model. MAPE-K is a feedback cycle for managing system adaptation [KC03]. iObserve extends the MAPE-K control loop with models shared between development and operations. These shared models carry architectural knowledge to ease the transition between development-level evolution and operation-level adaptation. The evolution activities are performed by human developers, while the adaption activities are executed automatically by predefined procedures, where possible, without human intervention.

The executed software system is observed to update architectural knowledge during operations. The model that reflects the architectural knowledge during operations is named architectural runtime model. As the architectural runtime model

is constructed by enriching and updating development models with operational observations, it is comprehensible for developers and operators and can be fed back into software evolution without the need of conversion and the risk of loss of knowledge. Each update leads to a new version of the architectural runtime model. Based on the up-to-date model, the current system configuration is analysed to reveal anomalies (e.g. increased usage intensity) and predict quality impact (e.g. upcoming performance bottlenecks). The architectural runtime model is then applied as input either for adaptation or evolution activities, depending on the outcome of a planning step. In the adaptation process, an adaptation plan is selected and evaluated to handle the anomalies. For adaptation planning, various design variants are created and evaluated on model level. Finally, the plan is executed to update the software system and its configuration. In the evolution process, changes are designed, evaluated and implemented by human developers.

The iObserve approach applies a mega-model to bridge the divergent levels of abstraction in architectural models used during development and operations. Mega-models describe the relationships of models, meta-models, and transformations [Fav04]. The iObserve mega-model depicted in Fig. 8.14 serves as an umbrella to integrate development models, code generation, monitoring, runtime model updates, as well as adaptation candidate generation and execution. Rectangles depict models and meta-models, respectively. Solid lines represent transformations between models, while diamonds indicate multiple input or output models of a transformation. Dots are used to indicate multiple input or output models of a transformation. Dashed lines reflect the conformance of a model to a meta-model and, in case of implementation artefacts, the instance of relationship between data and data types, for example the monitoring data and their corresponding event types in the instrumentation aspects, as depicted in Fig. 8.14.

The iObserve mega-model exhibits four sections defined by two dimensions: one for development vs. operations and one for model vs. implementation level. We discuss these four sections based on the CoCoME case study and the migration scenario.

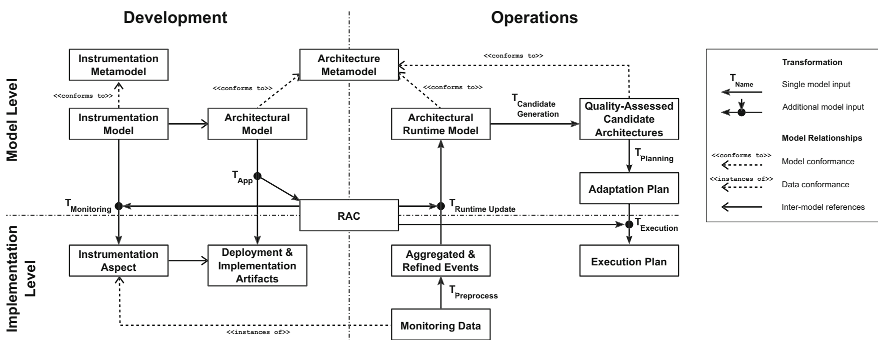

Fig. 8.14 Overview of the iObserve mega-model

For the interaction of the transformations in iObserve, we rely on the GECO approach [JHH16]. GECO defines patterns and methods to work with views and aspects on model and implementation level and describes how relationships between models and code can be shared between different transformations. These relationships are essential to map runtime observations to their corresponding runtime model elements, like classes and services, and design-time models to code artefacts. In iObserve, these relationships are created at design time with code generation or may be specified by hand in scenarios where the code is implemented by a developer. Subsequently, they are stored in the Runtime Architecture Correspondence Model (RAC). The RAC is the central element of the mega-model and is crucial for the use of an architectural model at development and operation time. At design time, they are used when generating and configuring the monitoring probes to map model-level pointcuts to implementation-level join points, select the correct probe technology, and probe introduction methods. At runtime, the same relationships are used in reverse to map runtime monitoring events, like an operation call, to their corresponding class and service instances.

Development Side On the development side at model level, the mega-model depicts the combination of an architectural model with our model-driven monitoring approach. We model the software architecture and deployment in a component-oriented fashion and generate the artefacts that are deployed and executed during operations. Therefore, iObserve relies on the Palladio Component Model [Reu+16] as an architecture description language defined through meta-models. The Palladio Component Model consists of several partial meta-models reflecting different architectural views on a software system. The monitoring part is specified using the instrumentation meta-model from our model-driven monitoring approach, consisting of two domain-specific languages used to describe monitoring events, for example operation calls, and the monitoring aspect [JHS13, JW16]. The aspect language allows to specify monitoring probes and their placements within the software system. For planning, we use probes to observe allocations/de-allocations, deployments/un-deployments, and user behaviour to learn the present system configuration and utilisation. The architectural and instrumentation models are then used to generate corresponding source code artefacts with the transformations T_{App}, for the software application, and the $T_{Monitoring}$, for the instrumentation [JW16].

At implementation level, the mega-model depicts development artefacts, including event types, instrumentation probes, and technology specific artefacts that implement the software system. For the CoCoME example, the software system is implemented by Enterprise Java Beans, and the monitoring uses Enterprise Java Beans interceptors to collect monitoring data.

Operations Side On the operation side at model level, monitoring data that adheres to source code artefacts is associated with the elements of the architectural runtime model. Consequently, the iObserve mega-model enables the reuse of development models during the operations stage by updating them based on operational observations. Moreover, the operation side shows the generation of adaptation candidate models and the adaptation plan construction. On implementation level,

a continuous stream of events is gathered by the monitoring probes. iObserve filters and aggregates the monitoring data ($T_{Preprocess}$), relates the monitoring data to architectural model elements, and finally uses the aggregated information to update the architectural runtime model ($T_{Runtime\ Update}$). Following the CoCoME example, increased usage intensity of the sale service triggers changes in the workload specification. $T_{Preprocess}$ filters out single-entry and -exit events of the sales service and aggregates them to sequences of events. Based on the sequences, the new usage intensity is calculated, which is then transformed to the architectural runtime model by $T_{Runtime\ Update}$. Therefore, the architectural runtime model connects the development and operation stages. It allows for stage-spanning consideration of software architecture. Furthermore, it enables quality analyses based on the architecture specification and is the basis for adaptation planning.

If a performance or privacy issue has been recognised, adaptation candidates are generated by transformation $T_{Candidate\ Generation}$ in the form of candidate architectural runtime models. These candidate models are generated based on a degree of freedom model that specifies variation points in the software architecture, which have been specified at design time. Once an adaptation candidate has been selected, the model is operationalised by deriving concrete tasks of a plan for adaptation execution. The tasks are derived by transformation $T_{Planning}$ while comparing a candidate model to the original model. The adaptation plan is transferred to an execution plan at implementation level by $T_{Execution}$.

For example in our CoCoME scenario, increasing utilisation of the sales service results in increasing response time. Performance forecasts indicate that the average response time of the service may exceed the performance SLA. Therefore, the deployment must be altered. Thus, various candidate models of the CoCoME architecture model are generated by $T_{Candidate\ Generation}$ each differing in deployment of the database service to data centres. The candidate models are analysed for quality. Once an appropriate candidate is found, the system is adapted based on the candidate model using $T_{Execution}$. Subsequently, the monitoring observes events that cover the deployment changes and updates the runtime model.

In case that no specific model among the candidates can be selected fully automatically, for example when there are trade-offs between quality aspects, or if an adaptation plan cannot be derived fully automatically, the human operator chooses among the presented adaptation alternatives. Also when no candidate model can be generated, for example due to lack of information or criticality of decision, the operator will be involved.

Summary In conclusion, this section describes how design-time architecture models can be used at runtime for performance forecasts and to generate candidate architectures used to steer the adaptation. These candidate models are, in essence, different versions of the running system with variations that are assessed for performance during candidate selection. As iObserve utilises the same model type for runtime and design time, the updated runtime models can be used during evolution and adaptation to assist performance forecasts and predictions, receptively. Thus, it keeps evolution and adaptation models aligned.

8.2.3 Co-evolution of Architecture and Analysis Models

As introduced previously, model-based performance evaluation is conducted using architectural and analytical performance models, as well as transformations among these models. In our work, we refer to both architectural and analytical performance models as quality models. When being constructed at design time, quality models are constructed from a system model, and it is assumed that the quality models reflect the system. However, architectural models can evolve in the life cycle of software, and this new version can lead to unexpected results if the quality model does not represent the system anymore. An example problem is the addition of a new software component without a corresponding addition of the state in the respective Markov chain. This inconsistency can lead to wrong performance analysis results. Hence, any version has to be realised as a co-evolution of all related design and quality models.

Handling this (co-)evolution is not a straightforward task [Get+18] because the quality evaluation model cannot be completely generated out of the system models, and most relations between the different models are not one to one.

We developed a framework called CoWolf [Get+15a] that is capable of incremental transformations. Therefore, it isolates changes that were done to a model to selectively propagate only these changes to the other models. In detail, the contribution of the CoWolf tool comprises mainly two aspects:

1. *The co-evolution of an associated model on the basis of model versions.* As described, models may have to be updated if other models changed. Often, those updates can be described canonically. CoWolf features the definition of rules that define the relation between model types. Using these rules, co-evolutions can be done (semi)-automatically for all associated models.
2. *Deliver utilities for model development and analysis.* For consistent development of the models, CoWolf provides a common environment with graphical and textual editors. Furthermore, it implements interfaces to external tools to analyse models.

Such tool is proposed to help system and performance viewpoint versions consistent during the evolution. First, the outcome of the performance analysis remains to present more accurate results. Second, different versions of the performance models are provided from the system model versions to be used for a possible incremental performance analysis.

In the remainder of this section, we are presenting the incremental transformations and co-evolution between sequence diagrams and Layered Queueing Networks (LQNs), as well as state charts and Markov chains. We show the relations between architectural models and various performance models for incremental changes. Accordingly, we discuss how a co-evolution on the performance models helps for performance analysis. A detailed and generic description of the CoWolf framework is provided by Getir et al. [Get+15a].

Sequence Diagram to Layered Queueing Network

We implemented the transformation of sequence diagrams to LQNs based on the description of Cortellessa et al. [CMI11]. They suggest a transformation from three source models as activity diagrams, component diagrams, and sequence diagrams to one target LQN model. The CoWolf co-evolution framework currently supports one-to-one transformations and aims to help the developer in the co-evolution of LQNs and Markov chains.

Since sequence diagrams do not include hardware information, the initial assumption is that multiple associated tasks are performed on one CPU. In the first step of every sequence diagram to LQN transformation, CoWolf checks if a CPU is part of the model and creates a new processor if it is missing. The user can increase the number of processors and change their properties, for example type and name, in the graphical editor. Each lifeline from the sequence diagram is transformed to a new task and a new entry type in the target LQN. We demonstrate the mapping elements, namely a new task and a new entry in Fig. 8.15. The new task is associated with the default processor created in the initial step.

All synchronous messages directed from a lifeline l_1 to a lifeline l_2 in a sequence diagram are mapped to exactly one synchronous call in the LQN. The source task of the call is the task mapped from lifeline l_1. The target task of the synchronous call is the task corresponding to lifeline l_2.

All asynchronous messages directed from a lifeline l_1 to a lifeline l_2 in a sequence diagram are mapped to exactly one asynchronous call in the LQN (see Fig. 8.16). The source task of the call is the task mapped from lifeline l_1. The target task of the asynchronous call is the task corresponding to lifeline l_2.

State Charts to Continuous Time Markov Chains

The transformations between state charts and Continuous Time Markov Chains (CTMCs) can be achieved via two-step transformations, namely state charts to Discrete Time Markov Chains (DTMCs) and DTMC to CTMC. These transformations

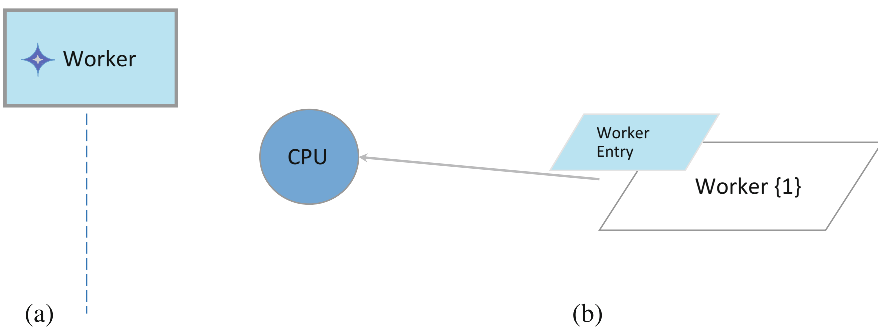

Fig. 8.15 Each lifeline will be transformed to a task. (**a**) Sequence diagram. (**b**) LQN

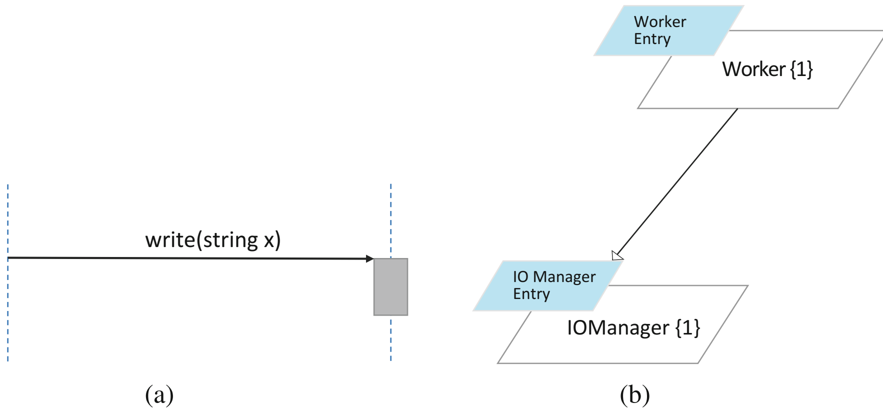

Fig. 8.16 All asynchronous messages between two lifelines (on direction) will be transformed to one asynchronous call in an LQN model. (**a**) Sequence diagram. (**b**) LQN

are implemented in a bidirectional way such that any change and any version can be reflected in both directions. The rules that are needed for the transformation between both model types are in all cases a simple bidirectional one-to-one mapping, where traces between elements of different diagram types could easily be created.

A CTMC is a common mathematical model to analyse software performance metrics like utility, throughput, etc. CTMC is a very similar but simpler model than PAAD, described in Sect. 8.1.3. However, this model is including more architectural information since it is an annotated form of an activity diagram.

The transformation is implemented only for the topmost states and transitions of a state chart model. States and transitions that are part of a composite state are not considered in the transformation. This does not match for `Action` elements (`Do`, `Entry`, `Exit`), which can call another sub-statemachine. For these actions, it is needed to find the first parent state that got transformed to DTMC and connect it by a transition to the initial state of the called sub-statemachine.

The transformation from DTMC to state chart has applied essentially the same transformation mappings. Additionally, some restrictions apply here:

- Created transitions in a DTMC always create a transition in the state chart model to avoid null name in the transformed model.
- Created states in DTMC always map to a state in the main sub-statemachine in the state chart model if there is no existing trace to a state elsewhere. It is not possible to transform a created state into another sub-statemachine.

After obtaining a DTMC from state charts, the user has to provide the probability distribution in the model. This looks a time-consuming task. However, assuming adding small information after many but small changes in the model, incremental transformations can be useful for the structural mapping and recommendations for

the co-evolution step. Unlike in a state chart model, transitions cannot be named in a DTMC model.

DTMC and CTMC are very similar models—especially structurally. Every CTMC state equates to a DTMC state, every CTMC transition equates to a DTMC transition, and every CTMC label equates to a DTMC label. The only difference is that CTMC states have an exit rate. The exit rate is calculated automatically from all outgoing transitions of a state. Furthermore, each transition t, outgoing from a state s, in a CTMC contains a rate. This rate is calculated as a fixed point by:

$$t.rate = t.prob * s.exitRate \tag{8.1}$$

Both models have the same elements. Changes can be directly applied from source to target models in both directions (CTMC to DTMC and DTMC to CTMC). The difference mapping is executed by a graph transformation rule.

Analysis of LQN and CTMC Models with Model Solvers

The analysis of an LQN model is performed by the LQN Solver [Car18]. In order to solve an LQN model with the LQN Solver, it is necessary to transform the model into a *.lqn* file.

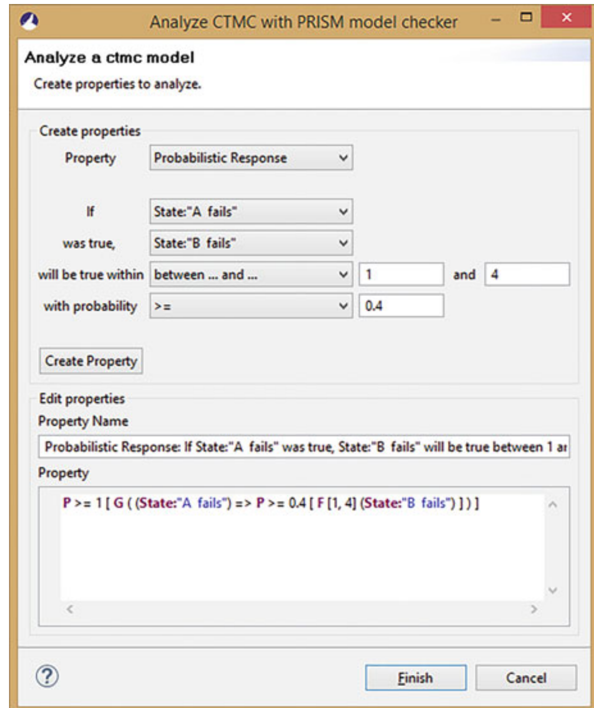

Fig. 8.17 CTMC properties wizard

DTMC and CTMC models are analysed using the PRISM model checker [KNP11]. CTMC models are used for performance and reliability analyses. Reliability can be validated using the reachability of critical states, for example error states. Performance can be validated in multiple ways. CoWolf provides a wizard (see Fig. 8.17) that helps to create default properties, which are "Steady State Probability", "Probabilistic Response", "Probabilistic Until", and "Probabilistic Existence". "Steady State Probability" calculates the probability that condition A will eventually become true. "Probabilistic Response" calculates the probability that condition B will always become true in a time frame after condition A was true. "Probabilistic Until" checks if condition A is always true before condition B becomes true. "Probabilistic Existence" checks if a condition becomes true in a time frame. As there are many more possibilities to evaluate CTMC models, additional properties can be created and edited in a text editor.

8.3 Conclusion and Road Map

Variants and versions of software challenge the measurement and prediction of performance. We illustrated selected strategies by means of two running examples, namely the PPU automation system and the service-oriented application CoCoME. The PPU comes with numerous variants, for which it does not scale to analyse each variant separately. We discussed solutions that reduce the variants that have to be measured and reduce the effort in assessing the performance for each variant by exploiting commonalities. CoCoME is an application that frequently evolves and requires to incorporate versions and different types of artefacts during performance analysis throughout the software life cycle. We discussed complementary solutions for performance analysis of software versions by making analysis techniques accessible and continuous by combining models and measurements.

In our experience, performance analysis of software does often not incorporate variants or versions at all. While all authors worked on improving the situation by explicitly supporting variants or versions, there is a research gap with respect to their combination. On the one hand, software being available in variants does indeed evolve over time, such that each variant exists in numerous versions. On the other hand, frequently evolving software is often also available in variants to tailor software to certain customers. However, techniques being used to address variants are often agnostic to versions, and techniques devoted to versions ignore the necessity to support variants. We envision that techniques for variants and versions are better integrated in the future but also that researchers focusing on variants can learn from research on versions and vice versa.

Performance analysis for variants can learn from research on versions. We and others have actively worked on sampling techniques to select a subset of variants that is sufficient to analyse. However, there is not a single sampling technique that incorporates the evolution of a product line. Furthermore, it is unclear to which extent currently available sampling techniques are stable (i.e. produce a largely

similar sample after evolution). Stability is especially important when assessing the performance evolution of variants, as different variants are likely to have different performance. Similarly, family-based techniques exploiting the commonality of a product line during performance analysis are typically oblivious to evolution. The encoding of the commonality of variants, however, may also be applied to encode the commonality of versions. This could lead to more efficient analyses for evolving product lines but may also be applied to versions of a software that does not come in several variants.

Performance analysis for versions can learn from research on variants. For versions, we have focused on the challenges associated with evolving and changing types of artefacts throughout the software life cycle. Possible areas in which the approaches from variants could be promising for versions is the efficient evaluation of design space and runtime reconfiguration alternatives—which essentially comprise variants of possible next versions. Another possible point of interaction emerges from modern software engineering paradigms such as DevOps and Continuous Software Engineering (CSE). In this context, new versions are created with an increasing velocity, multiple variants of a version are developed in parallel branches, and fast feedback about the quality is expected as part of the continuous delivery infrastructure and processes. This requires novel approaches for selecting and prioritizing performance analysis tasks, such as performance predictions or load tests. The sampling-based and family-based approaches for variants will be promising sources of knowledge.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.