Vergil: Guiding Developers Through Performance and Scalability Inferno

Christoph Heger*, Alexander Wert* and Roozbeh Farahbod†
*Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
Email: {christoph.heger, alexander.wert}@kit.edu
†SAP AG, 76131 Karlsruhe, Germany
Email: roozbeh.farahbod@sap.com

***Abstract*—Software performance problems, such as high response times and low throughput, are visible to end users and can have a significant impact on the user experience. Solving performance problems is an error-prone and time-consuming task that is ideally done with the help of experienced performance experts. They often provide solutions in the form of work activities to developers such as to move functionality from one component to another in order to solve performance problems. Existing approaches are mostly model-based and mainly neglect the code base and measurement-based techniques. They can miss important details from the implementation, configuration and deployment environment of the application. In this paper, we propose a novel approach in the field of software performance engineering with the goal to solve recurring performance and scalability problems based on a systematic process and formalization of expert knowledge. Starting with a set of detected performance problems in the target system, our proposed approach supports developers by identifying, evaluating and ranking of solutions, and by providing a work plan sketching the implementation of the selected solution. In an example with a Java EE application, we show the solution of a software bottleneck through result caching.**

Keywords*–*Software Performance; Software Engineering; Software Measurement; Performance Evaluation.

I. Introduction

Over the last decade, software performance practitioners have been documenting recurring performance problems and their identified root causes and solutions as performance anti-patterns (for example in [1][2]). The documentations include general definitions and solutions to the performance problems but nevertheless, solving recurring performance problems is still a manual and time-consuming task that requires expertise in software performance engineering [3], rigorous performance evaluation techniques [4], and a deep understanding of the system under study. After the presence of a performance problem has been observed and the root cause (or root causes) has been identified with the help of a performance expert, the solution process often includes a comprehensive analysis of the solution space and usually consists of (1) identification of possible changes, (2) evaluation of the performance impact of each possible change on the particular system, (3) effort estimation for applying one or more changes, and (4) deciding what changes to be applied to the system.

Currently, to the best of our knowledge, there are no approaches that help developers with the implementation of a performance and scalability solution with providing an ordered set of work activities at the code level. Existing approaches for solving performance problems, for example [5][6][7][8], are model-based. A shortcoming of model-based approaches is that not all performance problems can be solved at the model level [7] when the implementation, measurement-based experiments and monitoring-driven testing techniques are neglected. Additionally, cost factors and constraints have to be taken into account when a variety of solution choices exists. Furthermore, decision support mechanisms have to be integrated in order to support developers in selecting the most appropriate solution [9]. Only [8] considers using the effort estimation of the designer for the necessary design model changes in selecting a solution among alternatives. Jing Xu also uses the determined changes to suggest what should be changed in an abstract way, but not how to do it concretely [8]. Nevertheless, the existing approaches neither consider an existing code base, measurement-based testing techniques nor do they integrate decision support mechanisms or support the developer for implementing a solution at the code level.

In light of these observations, we are developing the Vergil approach (named after the ancient Roman poet Publius Vergilius Maro, Dante's guide through the inferno in The Divine Comedy [10]) that guides developers from a performance or scalability issue to solutions, by providing hypotheses about what to change, evaluating the changes in the context of the particular application and ranking the solutions to support developers in making a decision. Solution alternatives are provided as ordered lists of work activities sketching the implementation of the solutions for developers. Additionally, in order to support developers in making a decision on which solution to implement when different alternatives exist, it is necessary that developers estimate the effort to implement a certain solution. Therefore, developers often expect concrete work activities when asked for estimating the necessary effort. Vergil targets the development and maintenance phase of the software systems life cycle when an executable application implementation is available.

The core idea of Vergil is a process to identify and evaluate solutions to existing performance problems in a given application context, and to rank and recommend the most suitable of such solutions to the developers together with a description on how to implement them. The conceptual foundation is the formalization of performance expert knowledge into hypotheses about what to change and when. The goal of the approach is twofold: to make expert knowledge and methods for performance problem solution easily available to developers (who are not necessarily experts in performance engineering) and to guide them through the solution process with automation and tools [11].

In this paper, we introduce the overall concept of Vergil. The remainder of the paper provides an overview of Vergil's overall process, individual process activities, involved artifacts and how process activities and artifacts are connected.

In summary, we provide the following contributions in this paper:

1) We introduce Vergil's process, for exploring, evaluating and ranking of hypotheses to determine solutions of recurring performance and scalability problems.
2) We formalize performance problems as symptom traces in applications, and solutions as change hypotheses.

The remainder of the paper is structured as follows: In Section II, we introduce the process, its activities, and the artifacts of Vergil. In Section IV, we demonstrate the solution of a software bottleneck within a Java EE application through result caching. We present and discuss the related work in Section V and finally, conclude the paper in Section VI, also outlining our plan for future work.

II. The Vergil Approach

The main goal of Vergil is the provisioning of solutions (e.g., to split an interface or to move functionality to a certain component) to developers for solving performance and scalability problems. Vergil combines the strengths of a systematic process and the consideration of cost factors and constraints of model-based performance improvement approaches, for example [5][8][6], and extends them with the introduction of measurement-based performance problem solutions at the code level by means of monitoring-driven testing techniques, decision support mechanisms for selecting the most appropriate solution and work plans sketching the implementation of the solution.

There are two roles involved in Vergil as shown in Figure 1: Performance experts, who provide their knowledge about how to solve performance problems, and users (e.g., developers or other stakeholders; henceforth referred to as developers) who use Vergil to solve performance problems. The knowledge of performance experts about how to change a system is formalized in rules (henceforth called *Change Hypotheses*). Knowledge about how a change can propagate and impact other parts of the application is formalized in *Propagation Rules*. In each use case, developers provide the information about the problem context. They provide the *Performance Problem Model* by means of specifying the symptoms (e.g., high response times, high CPU utilization, or high memory utilization) and where they observe the symptoms in the application. They also provide the *Source Code* of the application, a *Test Environment* where the application can be deployed and the running application can be monitored during the execution of load tests, the *Performance Requirements* of the application as well as the willingness to change certain parts of the application, and constraints as *Developer's Preferences*.

Vergil uses all artifacts to test the applicability of *Change Hypotheses* and to evaluate which changes are leading to a performance improvement either with measurements and/or performance models. Vergil discards solutions that are not conforming with the *Developer's Preferences*. For each determined solution, Vergil derives a *Work Plan* with activities sketching the implementation of the solution. Developers estimate the implementation effort of each work plan. Vergil ranks the solutions based on all the collected information throughout

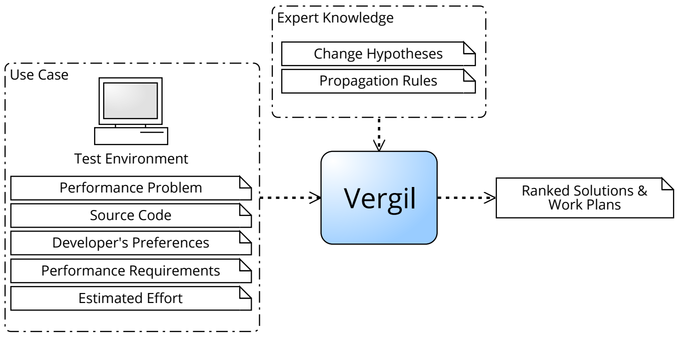

Figure 1: Vergil Overview.

the process and presents the ranked list as feedback to the developer. Developers can then discuss the solution proposals and implement the selected solution with the help of the *Work Plan*.

The process consists of four major activities as shown in the BPMN diagram [12] of Figure 2. In the context of this paper, we are focusing only on the overall concept of Vergil. The details of the activities *Propagate Work Activities* and *Estimate Effort of Work Plans* are described in [13].

A. Extract Models

The process starts with the *Extract Models* activity [14] that takes the source code of the application as input. The source code is parsed into the Source Code Model (SCM), for example with the Java Model Parser and Printer (JaMoPP) [15] for Java source code. An architecture performance model (APM) is extracted from the source code (in the context of this paper from Java) or when such a model already exists, it is imported. In the context of this paper, the APM is a Palladio Component Model (PCM) [16]. PCM is a software architecture simulation approach to analyze software at the model level for performance bottlenecks and scalability issues. It enables software architects to test and compare various design alternatives without the need to fully implement the application or buying expensive execution environments. PCM has already been used to detect and solve performance problems [7]. The PCM is created from the source code using the Software Model Extractor (SoMoX) [17]. The APM provides an architectural view of the application and is used to evaluate architectural change hypotheses in the remainder of the process. During the APM extraction, the Correspondence Model (COM) is build that links APM and SCM model elements, for example interfaces. A correspondence expresses the equality relation of two model elements in different meta-model instances. The SCM, APM, and COM are forwarded to the *Explore Change Hypotheses* sub-process.

B. Explore Change Hypotheses

The sub-process consists of the four activities *Test Change Hypotheses*, *Propagate Work Activities*, *Evaluate Work Activities*, and *Extract Work Plans*. Before we are going into the details of each activity in the remainder of this section, we introduce the performance problem model and our concept of change hypotheses. Change hypotheses provide the knowledge about what can be changed to solve a performance problem. The hypotheses are an important cornerstone in Vergil and are rules expressing what to change and when.

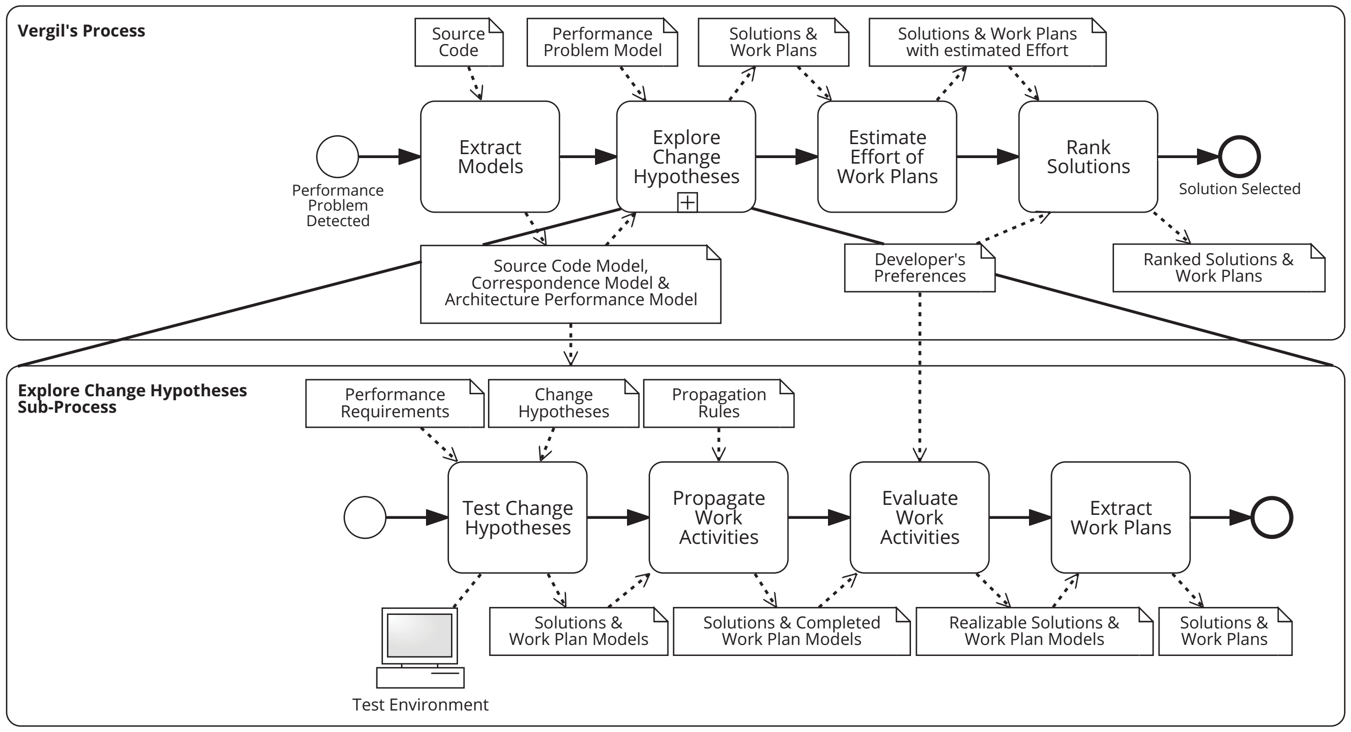
Vergil's Process
Source Code
Performance Problem Model
Solutions & Work Plans
Solutions & Work Plans with estimated Effort
Performance Problem Detected
Extract Models
Explore Change Hypotheses
Estimate Effort of Work Plans
Rank Solutions
Solution Selected
Source Code Model, Correspondence Model & Architecture Performance Model
Developer's Preferences
Ranked Solutions & Work Plans
Explore Change Hypotheses Sub-Process
Performance Requirements
Change Hypotheses
Propagation Rules
Test Change Hypotheses
Propagate Work Activities
Evaluate Work Activities
Extract Work Plans
Solutions & Work Plan Models
Solutions & Completed Work Plan Models
Realizable Solutions & Work Plan Models
Solutions & Work Plans
Test Environment

Figure 2: Vergil Process Overview.

Definition 1: A performance problem is a symptom trace through the application. It is formalized through a model of its root cause(s) expressed by its symptom(s), the workload specification [18] including the usage profile, and location(s) inside the application as shown in Figure 3. The resulting model is henceforth called *Performance Problem Model (PPM)*. A performance problem can have any number of other performance problems as cause expressed through the *causedBy* relation. One or more symptoms belong to a performance problem. A symptom can be among others: high CPU utilization, high response time, high memory utilization or high network utilization. A location is the referenced element of the SCM like a class, method, or statement where the symptoms are observed. The workload specification describes the workload (e.g., the number of users and their think time in a closed workload scenario) and the usage profile under which the symptom can be observed. The workload specification is formalized as a finite state machine and probabilistic usage behavior by means of Markov chains [18].

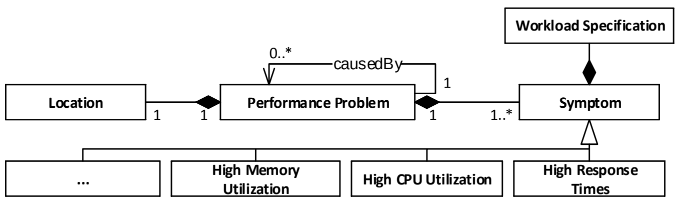

Figure 3: Performance Problem meta-model.

The PPM can be automatically extracted from a tool such as DynamicSpotter [19] or instantiated manually by the developer and is given as input to the process.

Definition 2: A change hypothesis h consists of a set of preconditions that must be fulfilled in order to be applicable in the problem context, a set of transformation rules that apply the changes of the hypothesis to the application on the defined level of abstraction (e.g., APM, SCM, etc.), a set of postconditions that test if the expected effect has taken place, and a work plan model template for creating the initial work plan model as shown in the meta-model in Figure 4. The conditions can test structural or behavioural properties of the application. A condition can consist of any number of structural (on the SCM, PPM and APM model) and behavioral (on measurement or prediction results) conditions testing static and dynamic requirements of the hypothesis. The conditions are rules expressed as logical predicates in first-order logic. First-order logic has already been used before in literature to formalize performance antipatterns [20]. The formalization of the changes depends on the level of abstraction. For example, in the case of APM and SCM (basically Java source code), graph rewriting rules (in place transformations) are used to transform the models. In the case of modifying parameter values in configuration files, simple text replacement rules are used.

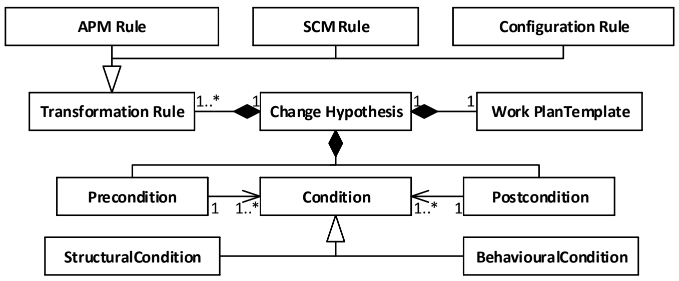

Figure 4: Change Hypothesis meta-model.

In the following, we provide an example of a hypothesis that caching the results of calling a method can improve performance:

The behavioural precondition of the hypothesis that caching the results of calling a method can improve performance ensures that the method m producing the results is deterministic. Deterministic means that for each method input $i \in \mathbb{I}$ and for all method calls c_m of method m there exists only one result $r \in \mathbb{R}$ in the set of results so that $c_m(i) = r$. The formalization of the precondition as one basic predicate *BP* is as follows:

$$\forall i \in \mathbb{I}, \forall c_m \in methodCalls(m), \exists! r \in \mathbb{R} : c_m(i) = r \quad (1)$$

where m denotes the method whose results shall be cached.

The structural precondition of the hypothesis matches a pattern in the SCM and PPM where a method is referenced from a performance problem with the "High Response Time" symptom and where the method implements an interface method. The structural precondition is formalized through the *BP* where m denotes the method (referenced by the performance problem) and m' denotes the interface method in the set of all methods:

$$\exists m, \exists m' \in Methods \subset SCM : \exists p \in PPM : \\ impl(m, m') \wedge ref(p, m) \wedge has(p, HighRespTimes) \quad (2)$$

The postcondition of the hypothesis ensures that the number of method calls of m has decreased after the changes of the hypothesis have been applied.

Taking the PCM as APM, the transformation rule of the hypothesis targets the PCM instance of the application. The performance-relevant behaviour of a method in PCM is modelled through a Service Effect Specification (SEFF) [16] (basically a series of actions) that can contain among others *BranchAction*, *InternalAction*, and *ExternalCallAction* elements. A *BranchAction* models a branch and can take the probabilities for each transition. An *ExternalCallAction* models the call to a method of another component. On an abstract level, caching results means that there is a probability P that the result is in the cache. This can be modeled in its simplest form by means of putting the *ExternalCallAction* EA, calling the method whose results shall be cached, inside a *BranchAction* that has the probabilities P for the cache hit transition and $1 - P$ for the cache miss transition (modeled through a *ProbabilisticBranchTransition*). Therefore, the rule simply wraps EA in the SEFF into a *BranchAction*.

The work plan model (WPM) template of the hypothesis is a blue print to create the initial work activities. A hypothesis knows the work activities resulting from the changes but not the possible side-effects. For example, a hypothesis to split an interface knows the work activity "Split". The work plan meta-model is introduced in the context of the *Extract Work Plans* activity in the remainder of the paper.

This concludes the hypothesis example. In the following, we describe the activities of the sub-process:

1) Test Change Hypotheses: The *Explore Change Hypotheses* sub-process starts with the *Test Change Hypotheses* activity that takes the change hypotheses H, the test environment, the performance requirements and the models as input. In this activity, the applicability of change hypotheses is tested, and the effect of the hypotheses' encapsulated changes is evaluated to build solutions. Jing Xu [8], Mauro Drago [21] and Diaz-Pace et al. [22] already considered performance evaluation of changes. The exploration algorithm selects sets of change hypotheses with fulfilled precondition and evaluates their effect through instantiating the changes in the context of the particular application and on the hypothesis' level of abstraction (e.g., architecture performance model, source code, or configuration file) and evaluates the performance. Vergil only considers changes that can be applied automatically. To give an example, two approaches for automated model refactoring for solving performance problems are presented in [23] and [8].

The set of change hypotheses H is the input for the *EXPLORE* procedure of the exploration algorithm as shown in Figure 5 (line 5). The current algorithm uses backtracking (as suggested by Arcelli and Cortellessa [9] and used in [24]) to find solutions that fulfill the performance requirements. The basic backtracking algorithm loops over all $h \in H \setminus H'$ and evaluates the precondition of h after the changes of the hypotheses in H' have been applied (line 6-7). If h_{PreCon} evaluates to $true$, the hypothesis h is evaluated. The postcondition of h is evaluated on the returned evaluation $result$ (line 8). When $h_{PostCon}$ evaluates to $true$ then h is added to H' (line 9-10). When the changes of a hypothesis are applied, the impacted elements of the application are identified as well as how they are impacted and also returned in $result$. The information is used to build the work plan models and its initial work activities based on the template. When $h_{PostCon}$ is not fulfilled, then the loop continues with the next hypothesis $h \in H \setminus H'$ (line 17). When the postcondition and the performance requirements are fulfilled, the hypotheses composition H' is added as solution to $solutions$ (line 12).

The performance requirements are expressed as upper bound of a performance metric. For example, the performance metric can be the response time of a method, the CPU, memory and/or network utilization of a server the application is running on. The performance evaluation and the postcondition also ensure that the changes do not lead to a performance degradation [9]. If the postcondition is fulfilled, but the performance requirements are not fulfilled, the procedure *EXPLORE* calls itself recursively with hypotheses composition H' and the set of hypotheses H (line 14). The result of the algorithm is the set $solutions$ that consists of sets of change hypotheses. Mathematically, the basic algorithm can miss solutions as it does not check all possible combinations of hypotheses (for practical reasons). However, a variation of exploration algorithm to test all possible compositions has to neglect the pre- and postcondition tests.

The performance improvement in Line 8 is either estimated by means of prediction through APM or determined through measurement-driven testing techniques on the System Under Test (SUT) as shown in Figure 6. The SUT consists of the deployed application and the *Test Environment (TE)*. The TE is a testing and monitoring environment where the application can be deployed and executed. Part of the TE is a load generator (e.g., HP LoadRunner [25], Apache JMeter [26]) and a representative load test for the application that is used to simulate users using the application. The load test itself consists of: a usage profile (how the application is actually used by its users), and the number of users to simulate and their think time (in a closed workload scenario). The usage

```
Set $H \leftarrow$ Change Hypotheses
Set $H' \leftarrow \emptyset$
Set $solutions \leftarrow \emptyset$

procedure EXPLORE(Set H', Set H)
    for all $h \in H \setminus H'$ do
        if evaluate($h_{PreCon}, H'$) then
            $result \leftarrow$ evaluate($h, H'$)
            if evaluate($h_{PostCon}, H'$) then
                $H' \leftarrow H' \cup \{h\}$
                if solved($result$) then
                    $solutions \leftarrow solutions \cup (H', result)$
                else
                    explore($H', H$)
                end if
            else
                Continue
            end if
        end if
    end for
end procedure
```

Figure 5: Exploration of Change Hypotheses.

profile is often a probabilistic behaviour. Given a currently visited web site, a user visits another web site or selects a certain element with a certain probability. The probabilistic usage profile and intensity-varying workload is specified in two types of models. A finite state machine specifies the possible interactions with the Web-based software system. Based on the finite state machine, the probabilistic usage is specified in corresponding user behavior models by means of Markov chains [18]. Markov4JMeter [27] implements such an approach for probabilistic workload generation by extending the workload generation tool JMeter. The probabilities can be determined from real-user monitoring of a deployed application running in production or manually through the expected usage of the application when no deployed application is available. In the latter case, a common, non-probabilistic usage profile is used.

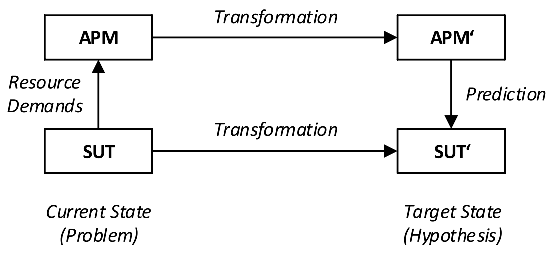

Figure 6: Evaluation Mean Alternatives.

For measurement-based experiments by means of monitoring-driven testing techniques on the SUT, our Adaptable Instrumentation and Monitoring (AIM) agents are deployed on the servers of the TE to instrument Java bytecode to monitor the application under load and to sample the resource utilization (CPU, network, etc.) of the servers. AIM provides means to automate the adaptation of instrumentation instructions. In experiment-based performance engineering, this feature can be utilized to automate a series of experiments to make manual interventions between individual experiments unnecessary. AIM specifies an extendable language to describe a desired instrumentation and monitoring state on an abstract level. It parses instances of the instrumentation description model and realizes instrumentation and monitoring instructions utilizing bytecode instrumentation, sampling and interception of the underlying Java Virtual Machine (JVM). A separate publication on AIM is in progress.

When the effect of changes is evaluated, the evaluation starts in *Current State* (cf. Figure 6) where a series of reference measurements S_0 is obtained. In the case of a $SUT \rightarrow SUT'$ evaluation, the measurements are obtained by means of monitoring-driven testing techniques with AIM and the execution of a workload. The source code transformation rules are applied to the SCM. The transformed SCM is used to transform the source code, for example in the case of Java, with JaMoPP. Configuration transformation rules are applied to the configuration files. Component configurations specified in the source code are treated as implementation transformation rules. The application of the transformation rules creates the SUT'. The series of evaluation measurements S_1 from the SUT' are then obtained analogous to S_0.

In the case of a $APM \rightarrow APM'$ evaluation, the resource demands for the calibration of the APM instance are obtained from the SUT. To determine the resource demands, the corresponding source code regions are instrumented with AIM to derive cumulative distribution functions $CDFs$ through experiment-based measurements with the SUT. The determined resource demands are inserted into the APM instance. The series of reference measurements S_0 is obtained through simulating the calibrated APM instance. The transformation rules are applied to the APM instance to create the APM'. In the *Target State*, APM' is simulated to derive the evaluation measurements S_1, as a prediction for the measurements expected from SUT'.

In both evaluation scenarios, the estimated performance improvement is determined as the difference $S_0 - S_1$. After the *Target State* is reached and S_1 is obtained, the changes are reverted (cf. Figure 5, line 8). The solutions and the corresponding WPMs fulfilling the requirements are forwarded to the *Propagate Work Activities* activity.

2) Propagate Work Activities: In this activity, the WPMs are completed by identifying all impacted elements of a solution and determining for any impacted element the required work activity. Vergil uses impact propagation rules to accomplish this task. The directly affected elements are identified during the instantiation of a change hypothesis (because it is applied to these elements). The WPM template of the change hypothesis provides the initial set of work activities. Changes can ripple through the application impacting other elements that are in a relationship (side-effects). To complete the WPMs, the side-effects and their work activities are determined through impact propagation rules [28]. Impact rules know if and how a work activity propagates itself to other elements, for example, when an interface in SCM is referenced from a "Split" work activity, then the rule knows that a class implementing that interface has to be splitted too. A side-effect can also be that new tests have to be added when a new interface is going to be created. Another example for such a follow-up activity is the redeployment of a component if the implementation will undergo changes. Rules are also used to conclude follow-up activities. The completed WPMs with the propagated impact and the corresponding solutions are forwarded to the *Evaluate Work Activities* activity.

3) Evaluate Work Activities: In the *Evaluate Work Activities* activity, the work activities are validated against the *Developer's Preferences*. Their referenced elements are tested if they can be changed. Developers express their willingness or unwillingness to execute a certain type of change on a grading scale. They can also express what cannot be changed, such as legacy parts of the application or the database. Arcelli and Cortellessa [9] already raised the concern that cost factors and constraints (e.g., the database cannot be changed) have to be taken into account when proposing solutions. Solutions whose WPMs contain unchangeable elements are discarded, removed from the set of solutions and the corresponding WPM is deleted. In the case of Java programs, Vergil takes unchangeable elements into account through the specification of the full qualified name or namespaces by the developers. For example, if a work activity references an element in the APM then an architecture impact is concluded. If the architecture is unchangeable, then the solution is discarded. The set of remaining solutions and WPMs are forwarded to the *Extract Work Plans* activity.

4) Extract Work Plans: In the *Extract Work Plans* activity, a list-based representation of the WPMs for the developers is extracted. The list of work activities sketches the implementation of the corresponding solution for developers. It also serves as foundation for the *Estimate Effort of Work Plans* activity. The list structure is determined through the *refinedBy* and *dependsOn* relations between work activities in the WPM. The *refinedBy* relation expresses the parent-child relationship of work activities whereas the *dependsOn* relation expresses the order of work activities.

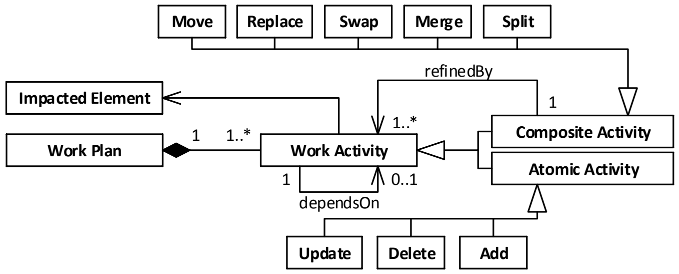

Figure 7: Work Plan meta-model.

Definition 3: A work plan is an ordered set of work activities. A work activity can be atomic such as add, delete, or update an element like a class, interface, and method or composite such as split, move, merge, swap, or replace elements [29]. A composite activity can be composed of other composite and atomic activities and is broken down until it is expressed through atomic activities. Refinement rules are used to break composite activities in the WPM down into atomic activities.

Work plans are not prescribing how the solution has to be concretely realized in the application. Jing Xu already motivated in [8] that the solution suggests what should be changed in an abstract way, but not how to do it concretely because there can be a host of ways. Vergil accomplishes this by modelling only abstract work activities sketching the implementation of a solution. The work plan can also, if necessary, list follow-up activities such as redeployment work activities and testing activities. The *Explore Change Hypotheses* sub-process ends after the extraction of the work plans is completed and forwards them together with the solutions to the *Estimate Effort of Work Plans* activity.

C. Estimate Effort of Work Plans

In the *Estimate Effort of Work Plans* activity, the effort for any work plan application is estimated by developers. This is a manual task done by the developers themselves because the effort can vary between individual developers depending on their knowledge, experience and practice. Vergil accepts the effort as unit less quantities for all atomic work activities. This leaves the decision of the concrete unit of measurement by the developers. Chosen once (in the current execution of the process), the unit of measurement has to remain the same for all work activities and work plans. The effort can be estimated, for example, in person (-hours, -days, or -months) [8]. The total effort estimation for a work plan is the computed sum of the unit less quantities of each atomic work activity. The consideration of the estimated effort takes the costs of solution alternatives into account [8], [9]. The solutions and the work plans with estimated effort are forwarded to the *Rank Solutions* activity.

D. Rank Solutions

In the *Rank Solutions* activity, the solutions are ranked through the rating of a multi-criteria decision analysis. The rating takes costs and constraints into account to support developers in deciding on an appropriate solution when a variety of choices exists [8][9]. The rating is done similar to [30] with a combination of the Analytic Hierarchy Process (AHP) [31] and the Simple Multi-Attribute Rating Technique (SMART) [32] taking the performance impact, cost factors, constraints and the developer's preferences into account. In the first step, AHP is used to obtain the priorities of the criteria. In pairwise comparisons, the developer judges the importance based on a fundamental scale of absolute numbers [31]. The priorities are given as input to SMART. SMART is a method of the multi-attribute utility theory. In contrast to the AHP where decision-making is done through pair-by-pair comparison of alternatives requiring human intervention, SMART ranks the solution alternatives based on the information already collected throughout the process using the given priorities (henceforth referred to as weights).

SMART uses a decision table consisting of m criteria $C_1, C_2, \ldots, C_m$ as rows and n solution alternatives $A_1, A_2, \ldots, A_n$ as columns. The cells contain the value of the alternative with respect to the criteria. Each criteria has an assigned weight w_i as dimensionless, normalized number originating from the developer's judgements (e.g., the importance of performance improvement, effort, or the willingness to change the architecture, etc.). For all alternatives, SMART computes the rating x_j of alternative A_j as follows:

$$x_j = \frac{\sum_{i=1}^{m}(w_i\ a_{ij})}{\sum_{i=1}^{m} w_i}, \quad j = 1, 2, \ldots, n \tag{3}$$

where a_{ij} is the normalized value of criteria C_i and alternative A_j.

The list of solutions is sorted descending according to the computed SMART ratings $x_1, x_2, \ldots, x_n$. The solution with the highest SMART rating in the list is placed on top.

Developers are then able to review and discuss the proposed solutions based on the work plans, the impacted elements—and how they are actually impacted, the costs, and the estimated performance improvement and to select a solution they are willing to implement. The selected solution and its work plan are the final result of the process.

III. Discussion

In this section, we clarify the current implementation status of Vergil's framework, the dependency on component-based software architectures and programming languages, and different categories of refactoring changes.

A. Automation of process activities

Currently, there are implementation prototypes for the two activities *Rank Solutions* and *Propagate Work Activities*. The activities of the *Explore Change Hypotheses* sub-process are intended to be automated in the near future. The *Estimate Effort of Work Plans* activity is not automated. However, Vergil still supports the developer by providing work plans sketching the implementation steps. In our next steps, we are designing the architecture of Vergil's framework based on the feedback we have received. Our goal is to design an architecture that allows tailoring the process to the specific needs of a use case. To give an example, in a certain use cause, it might be infeasible to estimate the implementation effort for each solution alternative. Instead, it is only feasible to estimate the implementation effort of the top-k solution candidates. Therefore, the solution alternatives must be ranked based on the criteria (neglecting the implementation effort) to identify the top-k solution candidates. In such a scenario, the rank solutions activity must occur twice in the process: (1) before the *Estimate Effort of Work Plans* activity, and (2) thereafter considering only the top-k candidates. In another use case, the developer may want to have all solution proposals in the ranking regardless of their conformity with the *Developer's Preferences*. In this case, the *Evaluate Work Activities* must be skipped.

B. Extension of the framework

Conceptually, Vergil is designed to be applicable to applications following component-based architecture and object-oriented design principles. The implementation of Vergil in the context of our research focuses on the Java programming language and the Palladio Component Model, which are both established means in industrial practice. We designed Vergil's framework to use exchangeable plugins to be able to support different programming languages and technologies. We specify and provide interfaces to implement plugins, e.g., to support the C# programming language. However, to make Vergil support other languages, there are certain key aspects that must be considered: programming language and technology specific knowledge encapsulated in Vergil's artifacts must be changed, extended, or developed to work with different languages and technologies.

C. Proposal of non-automatic evaluable solutions

In general, the changes of a change hypothesis can be assigned to one of the following three categories: (A) automatically executable, (S) semi-automatically executable, and (M) manually executable. Each category determines the ability to apply the changes of a change hypothesis automatically and the demand of human intervention, in order to evaluate the performance improvement of a change hypotheses (or a solution in general). The execution of refactorings in a work plan can also be categorized into (A), (S), and (M). As a result, we distinguish between nine possible categories of solutions described by the tuple:

$$Solution\ Category = Cat(DoA_{Ev}, DoA_{Ex}) \qquad (4)$$

where DoA_{Ev} determines the Degree of Automation (DoA) for evaluating the changes of a change hypothesis (the application of changes to the system respectively) and DoA_{Ex} determines the degree of automation for the execution of a work plan.

Categories (A, A), (A, S), and (A, M) are expected to require no human intervention to evaluate the performance improvement of a change hypotheses. Category (S, S) and (M, M), on the other hand, require human intervention. The category is often determined through the complexity of the refactoring. For example, simple refactorings like changing annotations are categorized as automatically executable. Refactorings categorized as semi-automatically or manually executable require the implementation of refactorings prior to their evaluation which is infeasible in most cases (cost vs. benefit trade-off). In order to avoid the implementation of changes just to evaluate the performance improvement, Vergil can still provide solution proposals in terms of work plans but without evaluating the performance improvement accompanied to the risk of introducing a performance degradation. Nevertheless, the proposal of such solutions can still be valuable for developers.

IV. MediaStore Example

In this section, we provide an outlook on the validation of Vergil. We present an excerpt of the *Test Change Hypotheses* activity by evaluating the change hypothesis given as example in Section II-B to cache the results of a method with the high response time symptom in a $APM \rightarrow APM'$ evaluation scenario. The example is structured as follows: In the current state, measurement-driven experimenting techniques are used to determine the resource demands from the SUT and to calibrate the *APM*. The calibrated *APM* is simulated to obtain the series of reference measurements S_0. Then, the hypothesis' changes are applied to transform the APM into APM'. The APM' is simulated to obtain the series of evaluation measurements S_1 (cf. Figure 6 and Section II-B1). To present preliminary results, we also show the response time measurements before and after implementing the changes in Figure 8.

We use a MediaStore [16] application as a simple use case example accessing the database and processing the data. The MediaStore allows its users to upload and store audio files as well as to download audio files encoded in a less or equal audio bit rate compared to the uploaded one. The application is implemented in Java EE and is deployed in a GlassFish 4 application server with Derby 10.10 as the database management system. The application server and the database management system are located on separate nodes. A short overview of the most relevant components for the example is shown in Figure 9 as excerpt from the PCM model. Only features relevant for the example are shown here and other

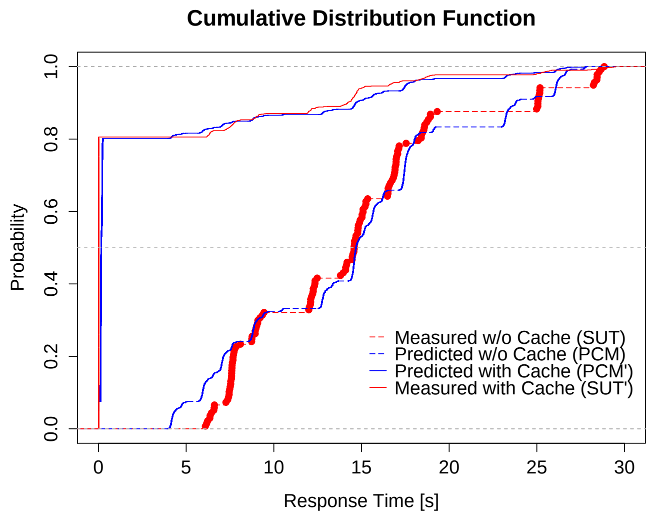

Figure 8: Measured and predicted response times.

PCM features can be found in [16]. We use the PCM model as APM in the example. The PCM model shows the resource container, on which the `WebGUIBean`, `MediaStoreBean`, `AudioAdapterBean`, and `EncoderBean` components are deployed. The resource container corresponds to the node in the *Test Environment* on which the MediaStore is deployed for measurement-based experiments. The SEFF models the performance-relevant behaviour of the `MediaStoreBean`'s download method and consists of the external call action to fetch an audio file from the database and the external call action to encode the audio file in a specified audio bit rate.

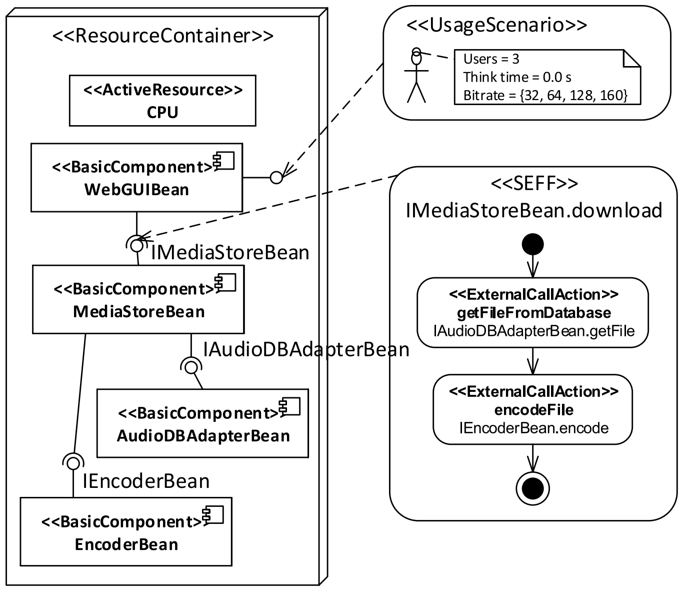

Figure 9: MediaStore PCM model excerpt.

We consider a scenario where multiple users download an audio file $\alpha \in AudioFiles, |AudioFiles| = 81$ randomly with a bit rate $\beta \in B = \{32, 64, 128, 160\}$ that is less compared to the uploaded bit rate of 190 kBit/s to force the re-encoding of α with bit rate β. Mathematically, the encoding function is defined as follows:

$$encode(\alpha, \beta) = \alpha' \quad (5)$$

where α' is the re-encoded audio file α in the desired bit rate β. The simulated usage profile is as follows: Users login, select the desired audio file α and bit rate β randomly following a uniform distribution, download the re-encoded audio file a', and logout. We simulate three power users with zero think time who execute the usage profile in a closed workload scenario using HP LoadRunner.

We instrument the `WebGUIBean`'s download method with our AIM agent and monitor the response time of the method. Therefore, the agent adds code statements at the beginning and the end of the method's body at the byte code level. The instrumentation (manipulating the byte code) is already fully automated. The added byte code instructions measure the time it takes to execute the method. In the monitoring results, shown as cumulative distribution function in Figure 8 (as dashed red line), we observe a measured median response time $\bar{r}_{mea} = 14.29s$ of the SUT in the applied workload and usage scenario. This is high in our considered scenario. The high response times are caused by the re-encoding of α.

We use the hypothesis (given as example in Section II-B) that caching the results of calling the `encode` method can improve performance. The `encode` method (as formalized in Equation 5) fulfills the precondition as it returns for the same input tuple (α, β) the same result α'. The size of an object cache is often specified by the number of elements that can be added to the cache before eviction takes place. In the case of a data access profile following a uniform distribution like in this example, the cache hit probability P only depends on the size of the cache and the total number of elements. For example, to achieve a hit probability $P = 0.8$, the cache size can be determined as follows:

$$\lceil |AudioFiles| * |B| * P \rceil = \lceil 81 * 4 * 0.8 \rceil = 260 \quad (6)$$

where the result is rounded to the next integer. In general, the size of the cache can be limited by the amount of memory that is available for caching objects. For the $APM \rightarrow APM'$ evaluation of the changes, we use the PCM model as shown in Figure 9, as APM. We extract the resource demands for the internal actions (not shown in Figure 9) in the SEFF of `IAudioDBAdapter.getFile` and `IEncoder.encode` with measurement-driven experiments on the SUT. The extraction is done (semi-) automatically. We are currently working on the full automation of resource demand extraction for PCM models with measurement-based experiments in the context of our publication about AIM. We calibrate the PCM model with the determined resource demands and simulate the usage profile and workload to obtain the series of reference measurements S_0. In S_0, also shown as cumulative distribution function in Figure 8 (as dashed blue line), we observe a predicted median response time $\bar{r}_{pre} = 14.39s$ in the current state of the APM.

In order to evaluate the hypothesis, we manually transform the SEFF `IMediaStoreBean.download` as shown in Figure 10. How the transformation can be automated is shown in literature [7][23][33]. We introduce a *BranchAction* and two *ProbabilisticBranchTransitions (PBT)* to simulate the cache. We assign the hit probability $P = 0.8$ to the *CacheHit* PBT and the miss probability $1-P = 0.2$ to the *CacheMiss* PBT. We assume the cache access time to be negligible, based on our practical experience (fetching an α' from the cache takes on average $0.02\mu s$ in the SUT' with the implemented caching solution).

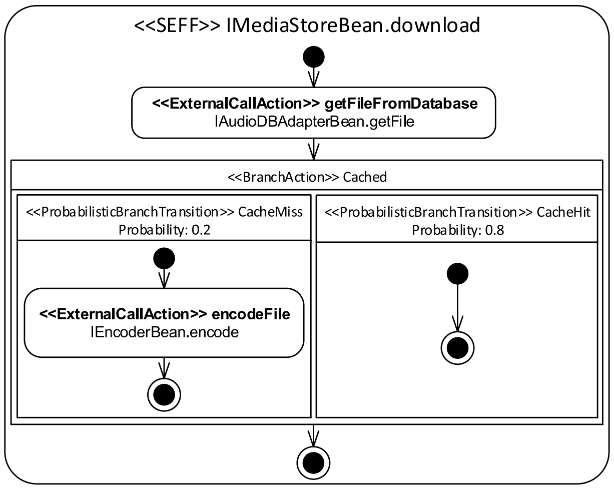
<<SEFF>> IMediaStoreBean.download
<<ExternalCallAction>> getFileFromDatabase
IAudioDBAdapterBean.getFile
<<BranchAction>> Cached
<<ProbabilisticBranchTransition>> CacheMiss
Probability: 0.2
<<ProbabilisticBranchTransition>> CacheHit
Probability: 0.8
<<ExternalCallAction>> encodeFile
IEncoderBean.encode

Figure 10: SEFF in target state of APM' with simulated cache.

The simulation results S_1 (denoted as blue line) for PCM' are shown in Figure 8 as cumulative distribution function. The simulation predicts a median response time $\bar{r}'_{pred} = 2.95s$. Based on the evaluation results, a performance improvement of 487% is estimated for the changes of the hypothesis.

To validate the simulation results, we also executed the $SUT \rightarrow SUT'$ evaluation. We implemented the cache as an Enterprise Java Bean with the help of Google's Guava libraries [34]. In the implementation of the `MediaStoreBean.download` method, the cache is checked first for the tuple (α, β). When the audio file cannot be obtained from the cache, α is fetched from the database and re-encoded with bit rate β. The resulting α' is added to the cache. We set the cache size to 260 objects and repeated the load test. The initial warm-up of the cache is done during the ramp-up phase of the load test. In the monitoring results (as shown in Figure 8 denoted as red line), we observe the measured median response time $\bar{r}'_{mea} = 2.71s$. The measured response times show a performance improvement of 527%.

V. RELATED WORK

The comprehensiveness of Vergil's process leads to a broad area of related fields of research. In the following, we cite only the most important and most relevant approaches to performance problem solutions due to space constraints. The interested reader may refer to [21][35] for more details about meta-heuristic approaches and generic design space exploration approaches. We categorize related approaches into model-based and measurement-based performance solution approaches.

A. Model-based Performance Solution

In [35], Cortellessa et al. present a model-based approach to automatically detect and solve performance antipatterns. Their approach targets the early design phase and the suggestion of architectural design changes to overcome performance problems. The goal of their proposed process is to modify a software system model to produce a new model without the performance problems of the former one [35]. They formalize antipatterns (often defined in natural language) as logical predicates in first-order logics [20][36]. Arcelli et al. present the automation of the model refactoring to improve the performance by applying model differences based on a Role-based Modeling Language [6][23]. Their approach suggests developers how to refactor models in order to remove problems. In the context of the approach of Cortellessa et al., Trubiani and Koziolek present the detection and rule-based solution of performance problems in Palladio Component Models in [7]. In [9], Arcelli and Cortellessa raise the need to take cost factors and constraints into account when a variety of solution choices exists and to integrate decision support mechanisms to support designers in selecting the most appropriate solution(s). In [8], Jing Xu presents a rule-based approach to detect and solve performance bottlenecks and long-path performance problems based on performance models. Models are modified with the help of the rules in ways that can be converted to design changes, which are then done manually. The costs for changing the design (carried out manually) is taken into consideration and can discourage rules from selecting changes, on a cost-effectiveness basis and for practical reasons. The proposed design changes describe what should be changed, and in what way, but not how to do it. The search is an iterative process. In each round, multiple alternatives can be created and the performance improvement is evaluated. When multiple design change branches are obtained at the end of each round, the performance improvement and weight of each branch is listed and ranked. Solutions are provided at the performance model level. Designers have to transfer the solutions from the performance model level to the design model level. In [37], Martens et al. propose an approach for automated performance improvement of component-based software systems based on meta-heuristic search techniques and rules applied to Palladio Component Models to find solutions for performance problems. In [21], Mauro Drago automates the detection-solution loop to automatically generate and propose design alternatives as feedback to the designer to improve non-functional properties of a software design. Quality-driven transformations are used to generate alternatives. Queuing Networks are used with estimated service demands to predict non-functional properties of each alternative. Diaz-Pace et al. [22] propose a framework to assist the software architect in the design of software architectures meeting quality requirements. Rules are used to change the design of the system. Currently, only rules to improve modifiability are supported that are applied to a graph-based representation of the architecture. The modifiability is evaluated with change impact analysis to determine the cost of changes while the performance is predicted with a simple performance model.

Neither of the approaches presented above considers an existing code base and measurement-based performance problem solutions nor do they support the developer in implementing the solution with an ordered list of work activities. The detection of performance problems with measurements and/or performance models is not in focus of Vergil. Also, Vergil targets the software development and maintenance phase of a systems lifecycle, when an implementation of the application is available. Vergil calibrates performance models with resource demands obtained from the system under test providing a more representative evaluation.

B. Measurement-based Performance Solution

Currently, to the best of our knowledge, there is no measurement-based performance solution approach that considers a comprehensive process for performance problem solution. In [14], Trevor Parsons uses monitoring-based techniques to extract a performance model of a Java EE application. The performance model is searched for detecting EJB-specific performance antipatterns. Problem solution is not part of the approach. To improve the deployment of components, Malek et al. introduce a framework [38] that guides the developer in the design of their solutions for component redeployment for large distributed systems. The goal is to find a deployment architecture that exhibits desirable system characteristics or satisfies a given set of constraints. They use runtime monitoring and consider quality of service (e.g., latency, availability). Aled Sage presents in [39] an approach for the observation-driven configuration of complex software systems. The author uses established statistical methods from manufacturing, called Taguchi Methods, and experiments to find configurations such as communication concurrency that meet the needs of various stakeholders. Lengauer and Mössenböck [40] propose the use of iterated local search methods to automatically compute application-specific Java garbage collector configurations. The selected configuration candidates are evaluated with monitoring-based techniques. The evaluation results are used to solve an objective function to determine the best configuration. In [41], Chen et al. use measurement-based experiments and source code changes in the context of object relational mapping only to prioritize the solution of performance problems based on the estimated performance improvement. However, their proposed approach does not consider performance problem solution.

Existing measurement-based approaches are focused mainly on a particular problem and its solution at the configuration level or at the architecture level. Neither of the approaches also considers the solution of performance problems at the code level nor do they provide a comprehensive process guiding the developer from a problem to a solution with work activities. Also, neither of the approaches consider cost factors and constraints for selecting the most appropriate solution when a variety of solution alternatives exist.

VI. Conclusion

Vergil guides developers from a detected performance or scalability problem to a solution. The proposed process explores hypotheses about what solutions can be applied to the software system, evaluates the performance improvement based on measurements and/or performance models, and ranks the solutions with respect to performance improvement, cost factors, constraints and the developer's preferences. The solutions are presented as an ordered list of work activities, sketching the implementation of the solution without prescribing to the developer how the solution is actually implemented. Strong concepts already used in existing model-based approaches are brought together and are extended with measurement-based performance problem solutions at the code level and the integration of decision support mechanisms to support the developer in selecting the most appropriate solution when a variety of choices exists. Vergil provides a comprehensive process for solving performance problems in the development and maintenance phase of an application's lifecycle where an implementation exists and where the solution of performance problems is known as expensive [42]. In this work, we presented the main idea, the details of the process and its activities as well as the formalization of performance problems and performance expert knowledge. Using an example, we presented promising preliminary results as a proof of concept for measurement-based performance problem solution and the calibration of performance models. We are currently working on the validation of the overall approach on a case study with a large open source e-commerce system. Additionally, we plan to conduct an empirical study with software performance consultants and developers.

Acknowledgment

The authors would like to thank Jonas Kunz and Sven Kohlhaas for their support that contributed to the paper. This work is supported by the German Research Foundation (DFG), grant RE 1674/6-1.

References

[1] C. Smith and L. Williams, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," in CMG-CONFERENCE-, 2003, pp. 717–725.

[2] B. Dudney, S. Asbury, J. Krozak, and K. Wittkopf, J2EE antipatterns. Wiley, 2003.

[3] C. U. Smith, "Performance engineering of software systems," Addison-Wesley, vol. 1, 1990, p. 990.

[4] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," ACM SIGPLAN Notices, vol. 42, no. 10, 2007, pp. 57–76.

[5] V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, and C. Trubiani, "Approaching the model-driven generation of feedback to remove software performance flaws," in Software Engineering and Advanced Applications, 2009. SEAA'09. 35th Euromicro Conference on. IEEE, 2009, pp. 162–169.

[6] D. Arcelli, V. Cortellessa, and C. Trubiani, "Antipattern-based model refactoring for software performance improvement," in Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures. ACM, 2012, pp. 33–42.

[7] C. Trubiani and A. Koziolek, "Detection and solution of software performance antipatterns in palladio architectural models." in ICPE, 2011, pp. 19–30.

[8] J. Xu, "Rule-based automatic software performance diagnosis and improvement," Performance Evaluation, vol. 69, no. 11, 2012, pp. 525–550.

[9] D. Arcelli and V. Cortellessa, "Software model refactoring based on performance analysis: better working on software or performance side?" in Proceedings 10th International Workshop on Formal Engineering Approaches to Software Components and Architectures, Rome, Italy, March 23, 2013, ser. Electronic Proceedings in Theoretical Computer Science, B. Buhnova, L. Happe, and J. Kofroň, Eds., vol. 108. Open Publishing Association, 2013, pp. 33–47.

[10] Virgil. [Online]. Available: http://en.wikipedia.org/wiki/Virgil [retrieved: 08, 2014]

[11] C. Heger, "Systematic guidance in solving performance and scalability problems," in WCOP '13: Proceedings of the 18th international doctoral symposium on Components and Architecture. New York, NY, USA: ACM, 2013, pp. 7–12.

[12] O. M. Group. Business process model and notation (bpmn). [Online]. Available: http://www.omg.org/spec/BPMN/2.0 [retrieved: 08, 2014]

[13] C. Heger and R. Heinrich, "Deriving work plans for solving performance and scalability problems," in EPEW. Springer, 2014, pp. 104–118, (in press).

[14] T. Parsons, "Automatic detection of performance design and deployment antipatterns in component based enterprise systems," Ph.D. dissertation, University College Dublin, 2007.

[15] Jamopp. [Online]. Available: http://www.jamopp.org [retrieved: 08, 2014]

[16] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," Journal of Systems and Software, vol. 82, no. 1, 2009, pp. 3–22.

[17] S. Becker, M. Hauck, M. Trifu, K. Krogmann, and J. Kofron, "Reverse engineering component models for quality predictions," in Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on. IEEE, 2010, pp. 194–197.

[18] A. Van Hoorn, M. Rohr, and W. Hasselbring, "Generating probabilistic and intensity-varying workload for web-based software systems," in Performance Evaluation: Metrics, Models and Benchmarks. Springer, 2008, pp. 124–143.

[19] A. Wert, J. Happe, and L. Happe, "Supporting swift reaction: Automatically uncovering performance problems by systematic experiments," in Proc. of the 35th ACM/IEEE Int'l Conference on Software Engineering, ser. ICSE '13. New York, NY, USA: ACM, 2013, pp. 552–561.

[20] V. Cortellessa, A. Di Marco, and C. Trubiani, "An approach for modeling and detecting software performance antipatterns based on first-order logics," Software and Systems Modeling, 2012, pp. 1–42.

[21] M. L. Drago, "Quality driven model transformations for feedback provisioning," Ph.D. dissertation, Italy, 2012.

[22] A. Diaz-Pace, H. Kim, L. Bass, P. Bianco, and F. Bachmann, "Integrating quality-attribute reasoning frameworks in the arche design assistant," in Quality of Software Architectures. Models and Architectures. Springer, 2008, pp. 171–188.

[23] D. Arcelli, V. Cortellessa, and D. Di Ruscio, "Applying model differences to automate performance-driven refactoring of software models," in Computer Performance Engineering. Springer, 2013, pp. 312–324.

[24] M. Drago, C. Ghezzi, and R. Mirandola, "A quality driven extension to the qvt-relations transformation language," Computer Science - Research and Development, 2011, pp. 1–20.

[25] HP LoadRunner. [Online]. Available: http://www.hp.com/go/loadrunner [retrieved: 08, 2014]

[26] JMeter. [Online]. Available: https://jmeter.apache.org [retrieved: 08, 2014]

[27] Markov4JMeter. [Online]. Available: http://www.se.informatik.uni-kiel.de/en/research/projects/markov4jmeter/ [retrieved: 08, 2014]

[28] S. Lehnert, Q. Farooq, and M. Riebisch, "Rule-based impact analysis for heterogeneous software artifacts," in Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on. IEEE, 2013, pp. 209–218.

[29] ——, "A taxonomy of change types and its application in software evolution," in Engineering of Computer Based Systems (ECBS), 2012 IEEE 19th International Conference and Workshops on, April 2012, pp. 98–107.

[30] F. Moges Kasie, "Combining simple multiple attribute rating technique and analytical hierarchy process for designing multi-criteria performance measurement framework," Global Journal of Researches In Engineering, vol. 13, no. 1, 2013.

[31] T. L. Saaty, "The analytic hierarchy and analytic network processes for the measurement of intangible criteria and for decision-making," in Multiple criteria decision analysis: state of the art surveys. Springer, 2005, pp. 345–405.

[32] W. Edwards, "How to use multiattribute utility measurement for social decisionmaking," Systems, Man and Cybernetics, IEEE Transactions on, vol. 7, no. 5, 1977, pp. 326–340.

[33] A. Koziolek, H. Koziolek, and R. Reussner, "Peropteryx: automated application of tactics in multi-objective software architecture optimization," in Proceedings of the joint ACM SIGSOFT conference (QoSA+ISARCS'11). New York, NY, USA: ACM, 2011, pp. 33–42.

[34] Google Guava-Libraries. [Online]. Available: http://code.google.com/p/guava-libraries/ [retrieved: 08, 2014]

[35] V. Cortellessa, A. Martens, R. Reussner, and C. Trubiani, "A process to effectively identify "guilty" performance antipatterns," in Fundamental Approaches to Software Engineering. Springer, 2010, pp. 368–382.

[36] V. Cortellessa, A. Di Marco, and C. Trubiani, "Performance antipatterns as logical predicates," in Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on. IEEE, 2010, pp. 146–156.

[37] A. Martens, H. Koziolek, S. Becker, and R. Reussner, "Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms," in Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering. New York, NY, USA: ACM, 2010, pp. 105–116.

[38] S. Malek, M. Mikic-Rakic, and N. Medvidovic, "An extensible framework for autonomic analysis and improvement of distributed deployment architectures," in Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, ser. WOSS '04. New York, NY, USA: ACM, 2004, pp. 95–99.

[39] A. Sage, "Observation-driven configuration of complex software systems," 2010.

[40] P. Lengauer and H. Mössenböck, "The taming of the shrew: Increasing performance by automatic parameter tuning for java garbage collectors," in Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, ser. ICPE '14. New York, NY, USA: ACM, 2014, pp. 111–122.

[41] T.-H. Chen et al., "Detecting performance anti-patterns for applications developed using object-relational mapping," in Proceedings of the 36th International Conference on Software Engineering, ICSE, 2014, pp. 1001–1012.

[42] B. W. Boehm, Software Engineering Economics, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.