TQPropRefiner: Interactive Comprehension and Refinement of Specifications on Transient Software Quality Properties

Sebastian Frank[1,2][0000−0002−3068−1172], Julian Brott[1][0000−0002−6253−3908],
Alireza Hakamian[2][0000−0001−9899−0062], and
André van Hoorn[1][0000−0003−2567−6077]

[1] University of Hamburg, Hamburg, Germany
`{sebastian.frank,andre.van.hoorn}@uni-hamburg.de`
[2] University of Stuttgart, Stuttgart, Germany

Abstract. Microservice-based systems are exposed to transient behavior caused, for example, by (frequent) deployments, failures, or self-adaption. The potential complexity of transient behavior scenarios makes specifying flawless transient behavior requirements challenging. Still, the required approaches and tooling to comprehend transient behavior and refine the requirements are lacking.

This paper aims to address this gap by providing a structured interactive approach that assists software architects in comprehending transient behavior and refining requirements. The prototypically implemented TQPropRefiner allows specifying transient behavior requirements using Property Specification Patterns (PSP). Then, TQPropRefiner uses runtime verification to evaluate requirement satisfaction on time-series data, e.g., from Chaos Experiments. TQPropRefiner visualizes the system's behavior and requirement satisfaction to foster comprehension. Based on the gathered insights, users can refine their requirements. In particular, TQPropRefiner currently supports refining timing constraints. Finally, we evaluated our early approach's feasibility and practical applicability in a qualitative user study with five industry experts. Despite currently limited support of PSP and refinement strategies, the preliminary results indicate that the approach can facilitate understanding transient behavior requirements among software architects and assist in the refinement process. Thus, our work is a first step toward facilitating the comprehension of transient behavior and refinement of requirements.

1 Introduction

In the last decade, major software companies have tended to deploy large applications in the cloud as small (micro-)services and benefited from greater agility, reduced complexity, and more effective application scaling in the cloud [24]. Due to their flexibility, microservice-based software systems are suitable for operating under frequent changes, e.g., load peaks, autoscaling, (re-)deployments, or failures. Changes in a software system usually temporarily affect the quality

This is the author's version of the work. It is posted for your personal use. Not for redistribution. The definitive version was published in *Companion of the 17th European Conference on Software Architecture (ECSA '23 Companion), September 18–22, 2023, Istanbul, Turkey*, 2023.

properties of a software system, e.g., response times increase due to a service failure. The term *transient behavior* denotes the system's behavior during the phase in which the system is not in a steady state.

It is important to make quality requirements and expectations regarding transient behavior explicit and to (in-)validate them [11]. For example, a too-long service recovery time may lead to customer frustration. Furthermore, disproving the expectation of the reaction time of an autoscaler can indicate severe problems in the system design and configuration. However, flawlessly specifying transient behavior is challenging. This is due to the complexity involved in the changes triggering transient behavior and the dynamic nature of transient behavior. One challenge is that specifying exact parameter values involves a lot of uncertainty among software architects, i.e., they often do not know whether their overall specification is feasible. Thus, learning from validating and refining the requirements is necessary. Approaches like Chaos Engineering [2] — building hypotheses and experimenting on the system to (in-)validate them — tackle this problem through an iterative refinement process. However, they are unspecific in guiding comprehension and refinement with strategies and methods. Furthermore, as shown in previous work [3], there is a general lack of proper tooling to address transient behavior.

This paper presents our approach to comprehension of transient behavior and refinement of transient behavior requirements. As a formalism for specifying requirements, we use Property Specification Patterns (PSP) [1] to transform human-readable Structured English Grammar (SEG) specifications into testable Metric Temporal Logic (MTL) [20] formulas. TQPropRefiner — the early prototypical implementation of our approach — guides software architects through the three steps of (i) specifying transient behavior requirements using PSP, (ii) validating the requirements against runtime data using runtime verification [22] and presenting the requirement satisfaction using visualizations, and (iii) refining the requirements by altering time constraints. Regarding refinement, our approach currently supports the use case in which a requirement is not satisfied and requires a timing parameter value modification to be satisfied.

We conducted an expert user study with five industry participants to gather early feedback on our approach and TQPropRefiner, despite limitations in the supported PSP and refinement strategies. The participants had to solve two tasks regarding comprehension and refinement capabilities of TQPropRefiner, answer a questionnaire, and participate in an interview. The participants were able to solve the tasks, and their answers indicate that our approach was easy to use. However, further improvements are necessary for use in practice, e.g., alignment of the shown visualizations and closer integration with system monitoring. Furthermore, the time constraint refinement needs more explanation.

In summary, the contributions of this paper comprise:

- An approach and tool (TQPropRefiner) that fosters comprehension of transient behavior to facilitate specification and refinement of transient behavior requirements. We make TQPropRefiner publicly available [6].

- Our vision and initial concept of refining transient behavior requirements. In particular, the implementation of a time constraint refinement strategy.
- The evaluation of our approach regarding feasibility and practicability in an expert user study. We provide the used documents and (anonymized) results as part of the supplementary material [12].

2 Background

2.1 Transient Behavior

Microservice-based software systems are usually complex and interdependent. Changes, e.g., failures, deployments, or self-adaptation, in one or more services may cause a system to transition from one steady state to another. This shift of states is described by the term transient behavior [4]. The concept of transient behavior originates from the field of electrical engineering. Within the state-space system model, there are two kinds of behavior: steady-state and transient. By performing transient analysis, it is possible to gain insight into the time-varying behavior of a system's Quality of Service (QoS) [25].

Since transient behavior is not focused on particular quality attributes and change types, it subsumes more specific concepts dealing with dynamic system behavior, e.g., survivability [15], elasticity [16], and resilience [21]. The quality of a system can be specified by quality requirements containing metrics such as response times. To identify occurrences of transient behavior, the actual QoS function of an underlying metric can be compared against the expected QoS [4]. Beck et al. [4] use Service-Level Objective (SLO) violations as indicators for transient behavior.

2.2 Property Specification Patterns

Transferring software system requirements to mathematical formulas to evaluate its quality can be challenging due to pragmatic barriers. To overcome this obstacle, Dwyer et al. [10] developed PSP to specify temporal logic formulas for recurring requirement scenarios. A PSP represents a generalized depiction of a frequently occurring requirement that governs the allowable sequences of events and states in a finite-state model of a system. Dwyer et al. [10] introduce the two pattern categories *order patterns* and *occurrence patterns*. Each pattern also has a scope, which defines an interval during the program execution in which the pattern must remain valid [10]. The scope is established by specifying the pattern's starting and ending state/event. Five different scopes exist: *Global*, *Before*, *After*, *Between*, and *After-Until*.

The initial PSP version is qualitative, i.e., it does not consider time constraints. To address this limitation, Konrad & Cheng [19] introduced Real-Time Specification Patterns. They describe these patterns as *quantitative* as they allow for quantitative reasoning about time. Such PSP can be mapped to MTL, among others, as done in this work. Autili et al. [1] further extend and align the available qualitative and quantitative patterns.

An example of an instance of the qualitative Response pattern is: *Globally, if {response time high} then in response {instance increase} eventually holds within 5 seconds*. In this example, *response time high* and *instance increase* are predicates, i.e., they evaluate to either true or false at specific points in time. The *5 seconds* is the time constraint on how fast the autoscaler must react.

3 Related Work

To our knowledge, only a limited number of approaches and tools holistically focus on specifying and comprehending transient behavior and refining transient behavior requirements.

The Property Specification Pattern Wizard (PSPWizard) [23,1] aims to simplify the selection and creation of PSP by providing a graphical user interface to construct supported patterns. A mapping generator allows the translation of the specified pattern into various target logics. The specification is not the core contribution of our approach, so we mostly reuse the concept of the PSPWizard. We further extend it by adding capabilities to specify predicates and visualize the satisfaction of predicates for the imported runtime data.

The Transient Behavior Verifier (TBV) [13] is a tool that provides an Application Programming Interface (API) for verifying transient behavior occurrences specified as PSP or MTL on monitoring data. The requirement satisfaction is visualized using a multi-line graph for the relevant metrics and colors to indicate requirement satisfaction over time. In our approach, we reuse TBV for its verification capabilities. Further, we reuse the visualization concept to show requirement satisfaction. However, we further extend the concept by also visualizing the satisfaction of the predicates involved in the requirement.

Hoxha et al. [17] developed VISPEC, a graphical tool for eliciting MTL requirements. VISPEC utilizes a graphical formalism automatically translated to MTL to assist non-experts in creating and visualizing formal specifications. Therefore, users can easily specify requirements without requiring training in formal logic. In that regard, we share the comprehension and visualization of temporal logic on runtime data. Nevertheless, VISPEC is focused on (initial) specification, while we focus on refinement of requirements. Furthermore, VISPEC uses an MTL-based graphical formalism in the specification process, while we use PSP and internally translate to MTL. Finally, our approach has a stronger focus on visualizing the satisfaction of requirements instead of supporting the specification.

The TransVis [4] approach assists software architects and DevOps engineers in specifying and evaluating transient behavior occurrences in their microservice systems. The tool displays the architecture of the assessed system and visualizes transient behavior in a graph. The user can interact with the tool via a chatbot, allowing for specifying simple requirements. The TransVis approach is based on the resilience triangle model from Bruneau et al. [7] in which transient behavior is characterized by the three indicators: initial loss of quality, time to recovery, and loss of resilience. Consequently, the specifications and visualizations are built

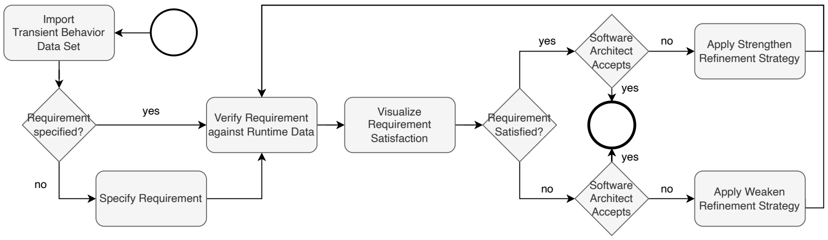


Fig. 1: Flowchart of the approach

specifically for these metrics, and there is no refinement assistance beyond visual comprehension. In contrast, we do not rely on the resilience triangle model and focus on requirement refinement.

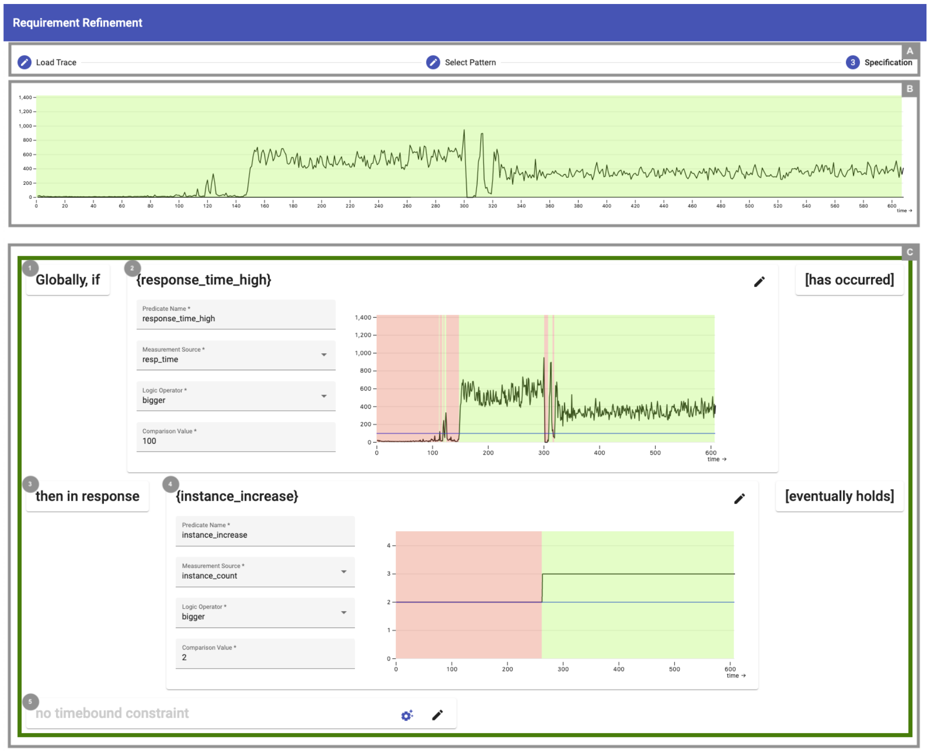
4 Approach and TQPropRefiner

4.1 Approach

The approach presented in the following is designed to assist software architects in comprehending and refining quality requirements in the context of transient behavior occurrences. Our underlying assumption is that transient behavior occurrences have been successfully identified, and data for a specified instance of transient behavior can be provided. Thus, our approach does not provide support for identifying transient behavior occurrences beyond visual inspection.

Our general approach is depicted in Figure 1. First, data from a detected transient behavior occurrence has to be imported. If not already available, an initial transient behavior requirement has to be specified. We use PSP as a formalism for these requirements since they are understandable to humans but also formal enough to be testable [9]. This property of TQPropRefiner is exploited in the next step, where we use runtime verification [22] to determine the satisfaction of (parts of) the requirement. Next, we visualize the runtime data and requirement satisfaction. Thus, software architects can easily decide whether the overall requirement is satisfied. Further, the software architect can consider the additional information to decide whether changes to the requirement are necessary, i.e., either because the specified requirement did not reflect the initial intention or new insights changed the expectation. A satisfied requirement can be strengthened to reflect new confidence in the system's capabilities. Vice versa, an unsatisfied requirement can be weakened to reflect the insight that the system behavior was actually good enough.

We introduce the concept of *refinement strategies* to transform a requirement into a refined one. Besides the actual transformation, a refinement strategy has the properties (i) *type*, (ii) *target*, and (iii) *assistance*. The *type* describes whether the strategy aims to strengthen or weaken (or both) a requirement. The *target* specifies which part of the PSP the transformation affects, i.e., the overall pattern, scope, predicate, or time bound. Finally, *assistance* describes whether the



1 Globally, if

2 {response_time_high}

Predicate Name *: response_time_high

Measurement Source *: resp_time

Logic Operator *: bigger

Comparison Value *: 100

[has occurred]

3 then in response

4 {instance_increase}

Predicate Name *: instance_increase

Measurement Source *: instance_count

Logic Operator *: bigger

Comparison Value *: 2

[eventually holds]

5 no timebound constraint

Fig. 2: TQPropRefiner showing (A) the step selection, (B) the pattern evaluation graph, and (C) the requirement specification & refinement

strategy actively assists the software architect in making a decision or whether it just shows the software architect the effects of already applied decisions. In this work, we focus on active assistance. An example of a strategy is “compute satisfying time constraint” (type: weaken, target: time bound, assistance: active), which we implemented as a first (active) strategy. In future work, we aim to add and investigate further strategies.

4.2 TQPropRefiner

Figure 2 shows the TQPropRefiner prototype in a state where (1) a data set (see DS_2 in Section 5) has been imported, (2) the *Response* PSP has been selected, and (3) an initial requirement (see T_2 in Section 5) has been entered. The tool guides the software architect through the three-step process of importing monitoring data, selecting a PSP, and specifying & refining the requirement. Each step can be accessed via the stepper component (see Figure 2 (A)).

Data Import The first step is to import a Comma-separated values (CSV) file containing time series data of monitored metrics, e.g., from a chaos experiment. The imported data is displayed in a table where each row represents the monitored data for each time unit, and the columns show the metrics.

Specification In step two, the software architect is asked to select a PSP as starting point for the initial specification. The selection is based on the pattern hierarchy introduced by Dwyer et al. [10] and the PSPWizard [23]. The software architect defines a scope, chooses a category (see Section 2.2), and finally picks a PSP. To provide additional context, the selected pattern is presented in the SEG as described by Autili et al. [1] and represented in a target logic of choice. However, only MTL is currently supported, and the pattern catalog is limited to three pattern variants: The *Response* pattern with the *Global* scope, and the *Universality* and *Absence* patterns with the *After* scope. We plan to add additional target logics and extend the supported patterns in the future.

The final step involves specifying the initial requirement and its refinement, as shown in Figure 2. To provide an intuitive specification process, the selected pattern is displayed as a SEG (see Figure 2 (C)). Each predicate of the pattern can be specified individually (see Figure 2 (C2) & (C4)). A predicate is specified by providing (1) a meaningful name, (2) selecting a measurement source (metric), which is populated from the imported data set, (3) selecting a logic operator, and (4) specifying a numeric comparison value.

Comprehension For verifying the PSP against the provided data set, TQPropRefiner uses the Transient Behavior Verifier [13]. We host an instance and access it via its API. The overall evaluation of the pattern is displayed in the graph at the top (see Figure 2 (B)). An all-green graph indicates the satisfaction of the entered requirement, while a red segment marks the moment the requirement is violated. The pattern evaluation result is also visualized by a green or red rectangle around the pattern (see Figure 2 (C)).

The predicates are individually verified against the provided data set, and the results are visualized in graphs. The time is represented on the X-axes, and the Y-axes represent the metrics. A selected metric is displayed in a black line chart, and the comparison value is a blue horizontal line. The time-dependent evaluation of the predicate is visualized by green segments for intervals the predicate is satisfied and red segments for unsatisfied intervals. In Figure 2 (C4), the specification of the *instance_increase* predicate is shown, which is defined as *instance_count* being greater than 2. The time-dependent evaluation of the predicate is visualized in the graph to the right. Corresponding to the specification, the interval of 2 instances being up is marked red, while the interval where the instance count increased to 3 is marked green.

Refinement To refine the pattern specification, the software architect can tweak its predicates. TQPropRefiner provides the passive refinement strategy of updating the visualization for the selected predicate and the overall pattern. This aims to facilitate a better comprehension of how changing one or more parameters affects the satisfaction of (parts of) the requirement.

For specifying and refining the time constraints, TQPropRefiner provides the implementation of an active refinement strategy (see Figure 2 (C5)). The tool performs a binary search based on the available predicate specifications to test potential time constraints. The resulting time-dependent verification result is displayed to the user showing for which time constraint intervals the pattern is satisfied following the same color coding we use for predicates. Currently, this refinement strategy is only available for time constraints, but we plan to add similar functionality for predicates in future versions of TQPropRefiner.

Implementation & Technologies TQPropRefiner has been implemented using the Angular framework in conjunction with the Angular Material UI component library. The code for the prototype is publicly available [6]. The modeling of PSP has been adopted from the PSPWizard [23]. We migrated the code to TypeScript classes, as the PSPWizard is implemented in Java.

5 Evaluation

To evaluate our approach's comprehension and refinement capabilities and practical applicability, we conducted a qualitative user study with five industry experts. We provided the experts with two tasks that needed to be solved using the prototype and asked them to evaluate their experience afterward. We investigate the following research questions:

- **RQ1:** To what extent can our approach facilitate comprehension of transient behavior occurrences among practitioners?
- **RQ2:** To what extent can our approach assist practitioners in refining requirements?
- **RQ3:** How can the approach be improved to assist practitioners in addressing practical challenges?

In the following, the provide details on our method, the provided tasks, the study execution, the results, and the discussion of the results and our method.

5.1 Method

We decided on a qualitative evaluation for two reasons. Firstly, the research questions focus on usability and improving an early concept and prototype. We argue that this can be best achieved by promoting a dialog with the study participants. This perception is supported by Greenberg & Buxton [14], who suggest that quantitative study designs could be detrimental in evaluating new ideas, particularly during prototype design, as they may limit expert feedback. Secondly, the complexity and the specialization of the covered topic lead to the practical barrier of finding enough participants to conduct a representative study.

We designed the expert user study not to exceed 1 hour and conducted it with each participant individually. In total, we gathered five participants,

Task 1 (T_1): Service Failure	**Task 2 (T_2): Load Peak**
Data Set 1 (DS_1)	Data Set 2 (DS_2)
– According to the SLO, response times may not exceed 150 time units – In the exceptional case of only 1 service being available, a response time of up to 400 time units is tolerated – In the experiment, 1 of in total 2 service instances has been terminated	– Response times may not exceed 100 time units. – In case the system is unable to satisfy the performance requirement, the number of instances should be increased – In the experiment, due to a load peak, service instances are scaled from 2 to 3
After {instances are smaller than 2}, it is never the case that {response times exceed 400 time units}.	*Globally, if {response times exceed 100 time units} then in response {the instance count increases to 3}.*
Is the requirement fulfilled?	How long did the system take to scale to 3 service instances?

Table 1: Context and SLO, initial requirement, and question for the two tasks

three working in a software company from the taxes domain and two working in a consulting and development company focusing on Application Performance Monitoring (APM). The participation did not demand any prior preparation.

5.2 Tasks

To solve the tasks, the participants received access to a hosted version of TQPropRefiner. We also provided two CSV files containing time-series data from two chaos experiments conducted by Frank et al. [13] with Chaos Toolkit (CTK) [8]. The first data set (DS_1) provided originates from *Chaos Experiment 1*, in which an injected fault caused a service instance to crash, leading to a response times increase. The second data set (DS_2) is from *Chaos Experiment 2*, in which the workload suddenly increases, and the implemented autoscaler is required to spawn an additional service instance.

Each task demands participants to go through four steps using TQPropRefiner. Firstly, each participant was asked to select a specific data set from a chaos experiment. Secondly, a suggested PSP from the pattern catalog needed to be selected. Thirdly, a given (initial) specification had to be entered by specifying the predicates of the selected PSP. Fourthly, a question on the requirement needed to be answered. Answering the questions may require the refinement of the initial specification. For each task, the participants have been provided with context information containing (i) the SLO of the underlying system defined by stakeholders, (ii) an initial specification, and (iii) a question as shown in Table 1.

We designed Task 1 (T_1) to evaluate to which degree participants are able to enter a given requirement specification and correctly interpret the verification result without any necessary refinement. Thus, T_1 is designed to address

RQ1. Task 2 (T_2) aims to evaluate to which degree participants can refine a given specification to examine a related requirement question. The answer to this question had to be derived from refining the time constraint of the selected specification. Therefore, T_2 addresses RQ1 and RQ2. To address RQ3, we conducted an interview with the participant to discuss potential improvements and required developments for practical use.

5.3 Execution

We conducted the evaluation online, with participants sharing their screens during the entire study. At the beginning of the session, we explained the study procedure. Afterward, we provided a link to a Google Form containing all information necessary for the study participation. This included seven questions on the participants' background knowledge, the two tasks to solve using TQPropRefiner, and 20 questions. The study host was present to answer potential questions from participants but did not actively intervene while the participants were going through the information on the Google Form.

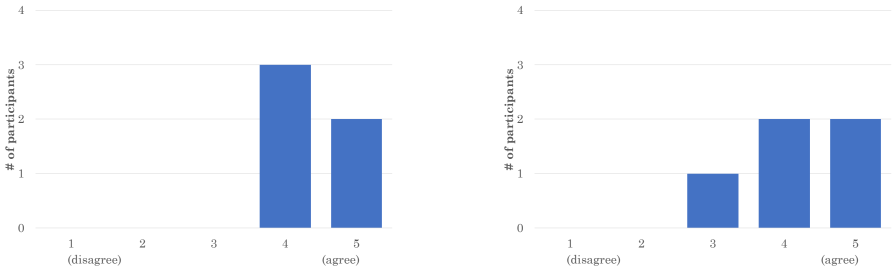
After solving the given tasks, we asked each participant to evaluate their experience concerning feasibility, usability, practical applicability, and potential improvements. To evaluate the prototype's feasibility, we asked the participants to rate their interaction with the tool on a Likert scale (one to five, one: strongly disagree; five: strongly agree). We based our useability questions on the System Usability Scale (SUS) [5] method. Finally, we gathered practical applicability evaluation and potential improvement suggestions using qualitative questions as well as a discussion between the participant and the session host.

5.4 Results

RQ1 No participant encountered problems entering the given specification into TQPropRefiner. Interpreting the evaluation result of a single predicate as well as the overall property was perceived as easy by all participants, who rated the comprehensibility for both with a median value of 4 out of 5. Additionally, all five participants were able to solve the tasks correctly.

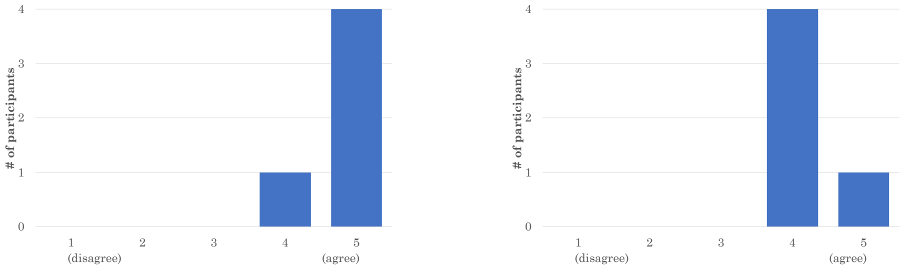
During the specification process, we observed that adding a time constraint to the requirement was not intuitive for some participants and, therefore, may require additional explanation within the tooling. Consequently, the answers to the ten SUS questions indicate that the tool overall was generally perceived as easy to use with a low entry barrier.

RQ2 All participants perceived the refinement of a single predicate as simple and rated it as easy with a 5 out of 5 median value. Refining the overall property was perceived as more difficult but was still rated with a median value of 4 out of 5. As part of the qualitative evaluation, we asked the participants whether they would have been able to solve the given task without TQPropRefiner. Two participants answered yes (they would just use the data visualization and manual inspection), two with no, and one with maybe. Also according to the results, T_2



(a) ... easy to interpret the initial evaluation of a single predicate.

(b) ... easy to interpret the initial property evaluation.



(c) ...easy to refine single predicate.

(d) ... easy to refine the property.

Fig. 3: Answers by the study participants for selected questions. It was...

has been solved correctly by four of the five participants. The wrong answer was due to the challenges of correctly interpreting the time constraint in the context of the overall pattern. However, the existence and the functionality of the tool's time constraint refinement feature were not intuitive to the participants. This needs better presentation and explanation in future versions.

RQ3 In open feedback, participants stated various ideas and requirements for potential production use of TQPropRefiner. Multiple participants pointed out that comprehensibility could be increased by horizontally aligning the predicate graphs. As depicted in Figure 2 (C2) & (C4), the two predicate graphs are not aligned, which makes identifying dependencies between various metrics difficult.

One participant elaborated that importing time series data as CSV files would be infeasible in production environments. Instead, an API integration of standard monitoring systems for trace import is required. For the question of whether the participants would frequently use the tool, the answers varied. Some participants agreed, but others pointed out that this depends on the precondition that they face tasks in their jobs where a tool like this would be beneficial.

Finally, participants provided some general potential improvements, e.g., adding a feature to save and load specifications, providing additional explanations on the color coding, and improving the tool's responsive design.

5.5 Discussion

The findings of RQ1 and RQ2 indicate that our approach is able to assist practitioners in comprehending and refining transient behavior requirements. The participants were able to enter a given specification, interpret verification results, and refine requirements. The tool's usability was perceived positively and has a low entry barrier. This is supported by the fact that the participants solved the given tasks by using the tool. Still, our approach must be improved, extended, and evaluated in a more exhaustive user study.

5.6 Threats to Validity

As a result of the evaluation, we have identified three validity concerns. Firstly, the group of participants was small and lacked heterogeneity. The five participants were employed at only two companies; some had similar expertise. Including software engineers without an APM background might have negatively impacted the results. Nevertheless, the number of (heterogenous) participants in qualitative studies is less critical. Studies with low (1 to 5) numbers of participants are not uncommon, according to Isenberg et al. [18].

Secondly, the tasks were designed specifically for the data sets we used for the evaluation. Since this data originates from academic experiments, they are not representative of the scenarios practitioners face in their production environments. Despite these concerns, we assume that the qualitative feedback we have received will be a first step in extending our early-stage prototype toward handling real-world challenges in the future.

Thirdly, some participants stated they could have solved the given tasks without TQPropRefiner. Thus, we must thoroughly investigate whether the comprehension and refinement of the requirement were facilitated due to using the tool, e.g., by comparing solutions obtained with and without TQPropRefiner.

6 Conclusion

This paper introduced our approach and tool TQPropRefiner for supporting software architects in comprehending transient behavior and refining requirements. In an expert user study, the participants were able to solve two tasks and confirmed the ease of use — providing evidence that our approach is a valuable step toward the interactive refinement of transient behavior requirements.

In future work, we aim to significantly extend the supported PSP, add support for more sophisticated predicates, and add more refinement strategies. In particular, we plan to extend the refinement to parameters involved in the predicates. Further, we aim to evaluate the approach in more realistic use cases and make the necessary improvements suggested by the participants, e.g., monitoring integration and alignment of the visualizations.

Acknowledgment The authors thank the German Federal Ministry of Education and Research (dqualizer FKZ: 01IS22007B and Software Campus 2.0 — Microproject: DiSpel, FKZ: 01IS17051) for supporting this work. The work was conducted in the context of the SPEC RG DevOps Performance Working Group[3].

References

1. Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. IEEE Transactions on Software Engineering **41**(7), 620–638 (2015)
2. Basiri, A., Behnam, N., Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., Rosenthal, C.: Chaos engineering. IEEE Software **33**, 1–1 (01 2016)
3. Beck, S., Frank, S., Hakamian, A., van Hoorn, A.: How is transient behavior addressed in practice? insights from a series of expert interviews. In: Companion of the 2022 ACM/SPEC International Conference on Performance Engineering. pp. 105–112 (2022)
4. Beck, S., Frank, S., Hakamian, A., Merino, L., van Hoorn, A.: Transvis: Using visualizations and chatbots for supporting transient behavior in microservice systems. In: 2021 Working Conference on Software Visualization (VISSOFT). pp. 65–75. IEEE (2021)
5. Brooke, J.: SUS-a quick and dirty usability scale. Usability Evaluation in Industry **189**(194), 4–7 (1996)
6. Brott, J.: Github project (2023), https://github.com/Cambio-Project/transient-behavior-requirement-refiner
7. Bruneau, M., Chang, S.E., Eguchi, R.T., Lee, G.C., O'Rourke, T.D., Reinhorn, A.M., Shinozuka, M., Tierney, K., Wallace, W.A., Von Winterfeldt, D.: A framework to quantitatively assess and enhance the seismic resilience of communities. Earthquake Spectra **19**(4), 733–752 (2003)
8. Chaos Toolkit Team: Chaos Toolkit (2023), https://chaostoolkit.org
9. Czepa, C., Zdun, U.: On the understandability of temporal properties formalized in linear temporal logic, property specification patterns and event processing language. IEEE Transactions on Software Engineering **46**(1), 100–112 (2018)
10. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Proceedings of the second workshop on Formal methods in software practice. pp. 7–15 (1998)
11. Eckhardt, J., Vogelsang, A., Fernández, D.M.: Are "non-functional" requirements really non-functional? an investigation of non-functional requirements in practice. In: Proceedings of the 38th International Conference on Software Engineering. pp. 832–842 (2016)
12. Frank, S., Brott, J., Hakamian, A., van Hoorn, A.: Supplementary material (2023), https://doi.org/10.5281/zenodo.8125612
13. Frank, S., Hakamian, A., Zahariev, D., van Hoorn, A.: Verifying transient behavior specifications in chaos engineering using metric temporal logic and property specification patterns. In: Companion of the 2023 ACM/SPEC International Conference on Performance Engineering. p. 319–326. ICPE '23 Companion, Association for Computing Machinery, New York, NY, USA (2023)

---

[3] https://research.spec.org/devopswg

14. Greenberg, S., Buxton, B.: Usability evaluation considered harmful (some of the time). In: Proceedings of the SIGCHI conference on Human factors in computing systems. pp. 111–120 (2008)
15. Heegaard, P.E., Trivedi, K.S.: Network survivability modeling. Computer Networks **53**(8), 1215–1234 (2009)
16. Herbst, N.R., Kounev, S., Reussner, R.: Elasticity in cloud computing: What it is, and what it is not. In: 10th international conference on autonomic computing (ICAC 13). pp. 23–27 (2013)
17. Hoxha, B., Mavridis, N., Fainekos, G.: Vispec: A graphical tool for elicitation of mtl requirements. In: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 3486–3492. IEEE (2015)
18. Isenberg, T., Isenberg, P., Chen, J., Sedlmair, M., Möller, T.: A systematic review on the practice of evaluating visualization. IEEE Transactions on Visualization and Computer Graphics **19**(12), 2818–2827 (2013)
19. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th International Conference on Software Engineering. pp. 372–381 (2005)
20. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-time systems **2**(4), 255–299 (1990)
21. Laprie, J.C.: From dependability to resilience. In: 38th IEEE/IFIP Int. Conf. on Dependable Systems and Networks. pp. G8–G9 (2008)
22. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming **78**(5), 293–303 (2009)
23. Lumpe, M., Meedeniya, I., Grunske, L.: Pspwizard: machine-assisted definition of temporal logical properties with specification patterns. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. pp. 468–471 (2011)
24. Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., Gil, S.: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: 2015 10th Computing Colombian Conference (10CCC). pp. 583–590. IEEE (2015)
25. Wang, C.Y., Logothetis, D., Trivedi, K.S., Viniotis, I.: Transient behavior of atm networks under overloads. In: Proceedings of IEEE INFOCOM'96. Conference on Computer Communications. vol. 3, pp. 978–985. IEEE (1996)