DiSpel Cockpit: Specification, Verification, and Refinement of Resilience Scenarios

Sebastian Frank[0000−0002−3068−1172], Aref El-Maarawi Tefur[0009−0005−7621−0746], Alireza Hakamian[0000−0001−9899−0062], and André van Hoorn[0000−0003−2567−6077]

University of Hamburg, Hamburg, Germany
{sebastian.frank,aref.el-maarawi.tefur}@uni-hamburg.de

Abstract. Chaos Engineering is an established method to assess the resilience of software systems by injecting failures and learning from experiments in production. Existing Chaos Engineering tools, such as Chaos Toolkit, facilitate creating and executing various failures but lack support for the entire process of resilience scenario elicitation, specification, execution, and refinement. This paper introduces DiSpel Cockpit for continuous and iterative specification, verification, and refinement of resilience scenarios. To achieve its goal, the DiSpel Cockpit combines the capabilities of existing tools into a holistic approach.
The DiSpel Cockpit uses Property Specification Patterns as a formalism to specify stimuli and responses of scenarios. System data is obtained from simulations and monitoring data. This paper presents the tool and demonstrates its usefulness based on resilience scenarios for an industrial system. We expect DiSpel Cockpit to assist software architects, particularly in the early phases of applying Chaos Engineering, when scenarios still have to be formalized, and feedback is necessary to gain confidence before moving toward conducting experiments in production.

We provide a video[1], source code, example data, and Docker containers[2].

1 Introduction

Resilience can be defined as a system's ability to continue its operation under adverse condition and to recover [12]. It is a quality that should be taken into account particularly when designing microservice-based software systems [13]. In the industry, Chaos Engineering (CE) [2] is an established method to build confidence in a system's resilience by injecting failures in production. It comprises an experimentation process in which hypotheses are stated and refined.

In previous work [7], we successfully designed and conducted a workshop to elicit resilience requirements in the form of quality scenarios as introduced by Bass et al. [3]. As the CE methodology suggests, we turned some of these

[1] Demo Video: https://youtu.be/gP6USBfOuxY
[2] Source Code & Releases: https://github.com/Cambio-Project/DiSpel-Cockpit

scenarios into chaos experiments. Although we were eventually successful, we also experienced challenges in the process. While plenty of tooling is available for conducting chaos experiments, e.g., Chaos Toolkit [4], we found the limited tool support for other activities in the CE process challenging. However, we also found conceptual challenges in the early phase of applying CE. CE as a method does not give explicit advice on how to (i) transform elicited resilience scenarios into experiments, (ii) get quick feedback on the feasibility of the scenario specifications, and (iii) refine resilience scenarios.

The DiSpel Cockpit implements the DiSpel approach [8], which aims to solve these issues. It provides a web-based graphical user interface and coordinates the extended and containerized tools PSPWizard [1], MiSim [10], MoSIM, TBVerifier [9], and TQPropRefiner [6]. Using our tool and PSPWizard [1], software architects can specify resilience scenarios using Property Specification Patterns (PSP) [1] to obtain testable scenarios. Next, relevant scenario occurrences can be obtained through simulation with MiSim [10] and search in monitoring data with MoSIM as a quick way to get feedback without setting up experiments. Our tool then performs runtime verification [11] with TBVerifier [9] to determine whether response specifications are satisfied. Finally, the scenarios can be refined by adjusting the parameter values in the provided PSP using TQPropRefiner [6].

In contrast to established CE tooling, like Chaos Toolkit [4], our approach is scenario-based, considers simulation and monitoring as data sources, and provides refinement strategies. Our previous work TQPropRefiner [6] is similar in some aspects but allows only specification of a single (response) PSP and excludes the detection of relevant occurrence data. Thus, the contribution of this work is a holistic, scenario-based CE tool focusing on the early phases of CE.

2 DiSpel Approach

In previous work, we introduced the vision of data-driven DiSpel approach [8], which proposes a continuous and iterative process for the specification, verification, and refinement of resilience scenarios as depicted in Figure 1. As such, the DiSpel approach is a derivative of CE [2]. In contrast to CE, the DiSpel approach considers a wider variety of data sources, i.e., besides chaos experiments, simulation, and monitoring data. Further, it introduces scenarios as a means to describe hypotheses, provides concrete refinement strategies, and allows analyses of transient behavior during experiments.

In the **specification** phase, software architects state hypotheses as scenarios [3], which consist of the elements stimulus, response, and environment, among others. PSP are used in the specification process for stimuli and responses. PSP [1] can be described as templates for common specifications. They can be represented in human-readable, Structured English Grammar (SEG) and translated into various testable, temporal logics.

In the **verification** phase, runtime verification [11] can be used to test against system data from various data sources since the scenarios are formally specified. The DiSpel approach considers *active* and *passive* data collection. Experiments

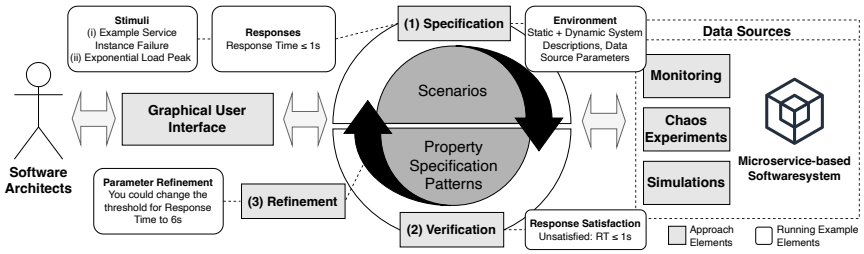

Fig. 1: Simplified depiction of the DiSpel Approach (adopted from [8])

and simulations can be actively triggered through the specifications to gather the data on the system's response. Simulations usually require modeling effort and lack precision but enable quick feedback and the potential to analyze what-if scenarios. As a passive and resource-efficient method, the specified incidents can be identified in monitoring data, if available. Note that all these methods will be applied based on the same (stimuli) specification.

In the **refinement** phase, practitioners can gain insights from the verification results on choosing feasible and appropriate resilience hypotheses for their system. By suggesting adjustments to parameter values of the response, such as a response time threshold, or modifying the entire pattern type of the response, practitioners can strengthen or weaken the response for a given scenario. This step is crucial due to the uncertainty in specifying exact values beforehand, as software architects often struggle to determine if their specifications are feasible.

3 Running Example

In a workshop, we elicited 12 resilience scenarios for a real payment accounting system under development, designed with a microservice-based architecture [7]. The sixth scenario describes an instance failure caused by a software bug and the third scenario an exponentially increasing load peak. As a running example, we synthesized a new scenario by combining these scenarios into a more complex scenario. This scenario reads as *immediately after an instance failure caused by a software bug, the number of wage clerks using the system rises exponentially during the payslip period at regular service hours. The system is expected to perform within a guaranteed tolerance, ensuring wage clerks always receive correct answers within 1 second (99% of the time)*. We use this example to demonstrate the tool's workflow and main features.

In our scenario, we identify two stimuli (instance failure and load peak) and one response (response time should be below 1 second, 99% of the time). Using PSP, we formalize the response by applying the Universality pattern as follows: *Globally, it is always the case that ResponseTimeOK holds.* Here, *ResponseTimeOK* is the system state more precisely described as *response time* $\leq$ *1 s*. Similarly, stimuli can be described through Existence and Response patterns. The full example is provided together with the tool. Once stimuli and responses

are specified using suitable PSPs, they can be mapped to temporal logics, e.g., Metric Temporal Logic (MTL) as described in the following:

$$\Box(\text{ResponseTimeOK}(\text{AllResponseTimes}))$$

We introduce the predicate ResponseTimeOK() with the AllResponseTimes metric as a parameter. The temporal operator $\Box$ means *always*. Thus, this expression evaluates to true when AllResponseTimes is less than 1 s for all its values.

4 DiSpel Cockpit

The DiSpel Cockpit serves as a unified platform that offers software architects a web-based user interface, which seamlessly integrates and coordinates existing tools, allowing our entire process to be managed and executed within the tool itself. We continue with the running example (Section 3 and Figure 1) to demonstrate how a user can apply the DiSpel approach (Section 2) using the DiSpel Cockpit. Excerpts from the user interface are displayed in Figure 2. We refer to the video and the example inputs on the project's GitHub page for details.

4.1 Specification

The user starts by creating a new resilience scenario on the *Scenario Editor* page. Creating a new scenario involves specifying the three main components of the scenario [3] format: the stimuli, the response, and the environment. The user can also provide a textual description of the scenario and a category tag for filtering and grouping related scenarios.

To set up the environment, the user uploads the necessary files, configuring the simulation and monitoring data retrieval processes implemented through MiSim [10] and MoSIM, respectively. The simulation requires an architectural model of the system under test, describing its architecture's static properties, like services, dependencies, and operations. Additionally, the user provides the experiment description and load profiles, which describe the dynamic properties of the system. For data retrieval from monitoring logs, users must supply monitoring data (currently a CSV file) and set a parameter for the MoSIM tool to determine the duration of the occurrences extracted from the monitoring data.

The stimuli and responses are specified using PSP. The *PSP Editor* page facilitates the creation of specifications through a UI. Users construct the specification by selecting the scope and appropriate pattern type, followed by the pattern itself and any additional building blocks compatible with the chosen pattern. Figure 2 (A) shows a part of the UI after specifying the running example's response. In the background, the PSPWizard [1] translates the specified pattern into the selected target logic, and the results are displayed and persisted.

4.2 Verification

Figure 2 (C) shows the compact presentation of a fully specified scenario in the *Scenario Overview*, a list of all specified scenarios. The user can initiate the verification process by running simulations and searches over monitoring data. The

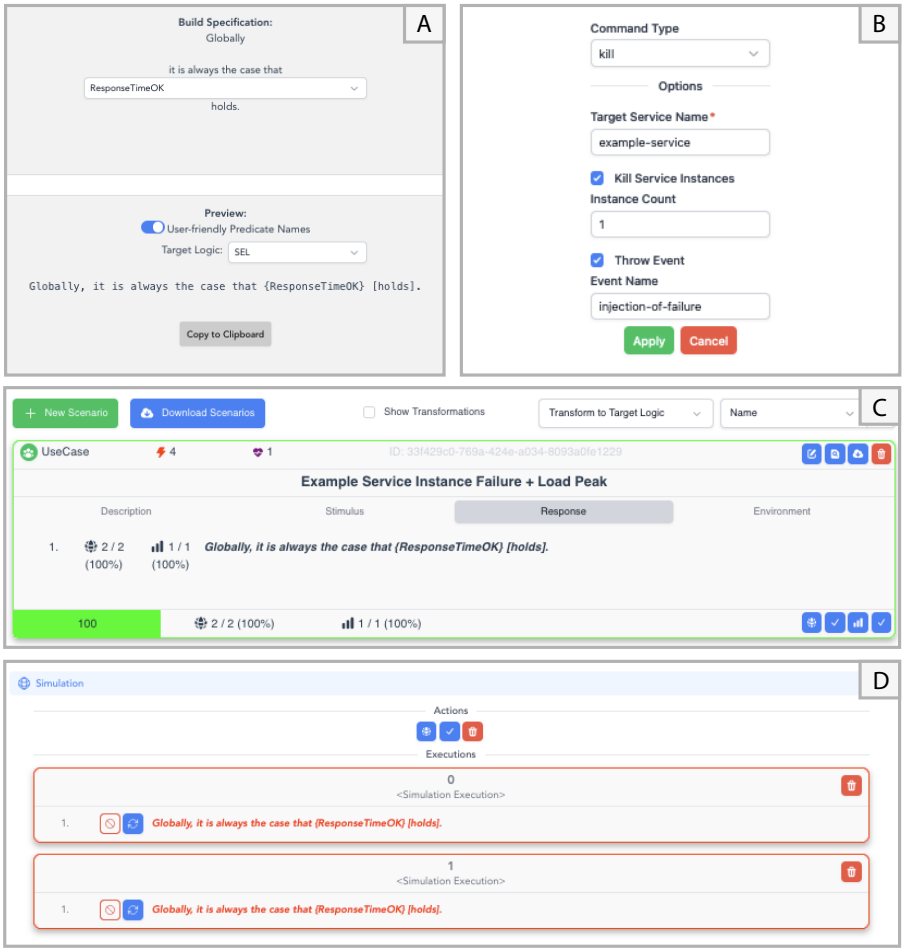

Fig. 2: Selected screenshots of the DiSpel Cockpit UI

specification of stimuli and responses is fundamentally similar, particularly regarding the pattern selection. However, responses contain only events derivable from system metrics. In contrast, stimuli can (and should) also contain commands because stimuli serve as the instructions for retrieving relevant system data. The DiSpel Cockpit assists the software architect in formulating listenable events and commands by providing wizards. Figure 2 (B) shows the wizard for specifying the *kill* command *killExampleServiceInstance*. This command can then be used in a stimulus specification using the Existence pattern: *Globally, (killExampleServiceInstance) [holds] eventually between 20 and 20 time units.*

Note that once a command is executed, the simulator can trigger a listenable event. After the *killExample-Service-Instance* command is executed, it emits the listenable event *injection-of-failure* (see Figure 2 (B)). In further stimuli

specifications, this event acts as a trigger to induce a workload peak. Currently, only *kill* and *load* commands are available. For monitoring data search, using the same stimuli specifications, commands must be substituted by equivalent events. MoSIM provides default implementations in an early state to achieve this. For example, instead of terminating a service instance, the system searches the monitoring data for a drop in the service's instance count as a heuristic.

Responses are later in the process used to test against the retrieved system data, which is performed by TBVerifier [9]. As shown in Figure 2 (D), the Dispel Cockpit displays the verification results using color cues. The red color coding indicates that neither the whole scenario nor the individual responses were satisfied for both simulation runs. Further, metrics are provided that show the scenario's overall resilience score and the share of satisfied scenario occurrences from simulation and monitoring (see bottom of Figure 2 (C)).

4.3 Refinement

If the response specification is not satisfied for a particular stimulus occurrence, the Dispel Cockpit aids the user in investigating the cause and refining the specification. The *Refinement View*, powered by TQPropRefiner [6], visualizes the system's behavior and the requirement satisfaction. Interactive refinement strategies assist software architects in making informed decisions and understanding the impacts of their choices. For instance, if the response time in our example scenario exceeds the specified 1 s threshold, the tool tests and suggests alternative threshold values. A possible refinement would be to weaken the response time threshold by setting it to a higher value. Once potential refinements are identified and accepted, a new cycle of verification and re-refinement can begin.

5 Implementation

The DiSpel Cockpit is a web-based application following a client-server model. The frontend is a single-page application built with the lightweight Nuxt.js, a Vue.js-based framework. In contrast to Vue, Nuxt enables backend development support, allowing communication between the frontend and backend services. MongoDB is responsible for persisting the scenarios and analysis metrics. System data is currently stored in a shared Docker volume.

The Cockpit's backend is structured as microservices architecture, developed using different programming languages (Java, Kotlin, Python) and frameworks (Angular), integrating the five tools PSPWizard [1], MiSim [10], MoSIM, TBVerifier [9], and TQPropRefiner [6] as services. These services run in Docker containers orchestrated using Docker Compose, enabling independent deployment and enhancing flexibility and scalability.

Most backend services previously lacked APIs and provided GUIs only. For PSPWizard and MiSim, we implemented REST APIs by encapsulating the API functionality within a Java submodule using the Spring Boot framework to retain these tools' standalone capability. TQPropRefiner required adaptions of

functionality, GUI components, and its API for integration into our Vue frontend. We established the TQPropRefiner as a separate service, embedding its GUI elements within the Vue.js frontend as a web page. This facilitated seamless integration without extensive code modification. Some (functional) service changes were necessary to allow for proper interaction between the tools. Among others, we added a modern UI and a new target language to the PSPWizard to match the syntax expected by TBVerifier and extended TQPropRefiner to generate its GUI dynamically for support of various PSP types.

6 Discussion

During the development of the DiSpel Cockpit and its underlying tools, we have been in exchange with an industry partner to ensure the tool's usefulness for practical use cases. In early feedback, practitioners praised the expressiveness of the PSP-based resilience scenarios and the capabilities for analyzing transient behavior. In addition, the running example used during development is based on real scenarios [7], and the simulation model has been reused from previous work [10]. Further, PSP have shown to be sufficient to capture quality requirements in industrial case studies [1]. With limited generalizability, this indicates the feasibility of our approach. Nevertheless, a more thorough and systematic evaluation of the tool's capabilities to aid software architects in the early phases of CE is still necessary.

While there is no evaluation of the DiSpel approach and tool as such, specific aspects and underlying tools have been partially evaluated before [6,10,9]. For example, expert users solving tasks with TQPropRefiner were mostly successful and found it easy to refine specifications [6]. Further, Czepa and Zdun [5] have shown the understandability of PSP be superior to plain temporal logic.

The DiSpel Cockpit currently suffers from technical and conceptual limitations, both partially originating from the underlying tools. While conceptually compatible with PSP, the tooling's implementation currently does not support composed and complex predicates (like *Service 1 fails and Service 2 fails*) and the handling of time units. It also requires equally sized time steps in the analyzed data. Further, the supported command (currently: kill and load), listener (currently: user defined events), and PSP types must be extended. Regarding the conceptual limitations, early user feedback suggests collecting monitoring data from monitoring systems, executing actual chaos experiments, and adding support in eliciting and (graphically) specifying scenarios would increase the tool's utility. By addressing these limitations, we plan to extend the work to support more (realistic) scenarios and industrial, real-world systems.

7 Conclusion

In this paper, we presented the DiSpel Cockpit, a tool that allows software architects to specify, verify, and refine resilience scenarios. To reach this goal, the DiSpel Cockpit leverages the capabilities of existing tools, i.e., the PSPWizard

for specifying PSP, the resilience simulator MiSim and the stimuli search library MoSIM for generating/finding data, the TBVerifier for analyzing scenario satisfaction, and the TQPropRefiner for refining scenarios.

Although we provided an essential proof-of-concept of the DiSpel approach through the DiSpel Cockpit, a thorough evaluation of its usefulness and presumed benefits in a more realistic setting is still necessary, e.g., in user studies. In future work, we also intend to improve and extend the connected tools and the DiSpel Cockpit itself, e.g., by adding transformations to chaos experiments, employing graphical specification, and assisting in eliciting scenarios.

Acknowledgments. The authors thank Angelina Heinrichs, Marvin Taube, Alexander Baur, Patrick Mohr and Julian Brott for contributions to the tool and the German Federal Ministry of Education and Research (dqualizer FKZ: 01IS22007B and Software Campus 2.0 — Microproject: DiSpel, FKZ: 01IS17051) for supporting this work.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

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. Bass, L., Clements, P., Kazman, R.: Software architecture in practice. Addison-Wesley Professional, 4 edn. (2021)
4. Chaos Toolkit Team: Chaos Toolkit (2023), https://chaostoolkit.org
5. 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)
6. Frank, S., Brott, J., Hakamian, A., van Hoorn, A.: TQPropRefiner: Interactive comprehension and refinement of specifications on transient software quality properties. In: ECSA'23 Post-Proceedings (2023), (in press)
7. Frank, S., Hakamian, A., Wagner, L., Kesim, D., Zorn, C., von Kistowski, J., van Hoorn, A.: Interactive elicitation of resilience scenarios based on hazard analysis techniques. In: ECSA'21 Post-Proceedings. pp. 229–253. Springer (2021)
8. Frank, S., Hakamian, A., Wagner, L., von Kistowski, J., van Hoorn, A.: Towards continuous and data-driven specification and verification of resilience scenarios. In: ISSREW'22. pp. 136–137. IEEE (2022)
9. 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: ICPE'23 Companion. p. 319–326. ACM (2023)
10. Frank, S., Wagner, L., Hakamian, A., Straesser, M., van Hoorn, A.: MiSim: A simulator for resilience assessment of microservice-based architectures. In: QRS'22. pp. 1014–1025. IEEE (2022)
11. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming **78**(5), 293–303 (2009)
12. National Institute of Standards and Technology: NIST SP 800-39 (2011)
13. Newman, S.: Building Microservices. O'Reilly Media (2015)