Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Supporting and Verifying Transient Behavior Specifications in Chaos Engineering

Denis Zahariev

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn
Supervisor:	Dr.-Ing. André van Hoorn, Sebastian Frank, M.Sc. Mir Alireza Hakamian, M.Sc.
Commenced:	January 28, 2022
Completed:	July 28, 2022

Abstract

Context. Chaos Engineering is an approach for investigating the resilience of software systems, i.e. their ability to withstand unexpected events, adapt accordingly, and continue providing functionality. An integral part of the approach is continuous experimentation, expressed in continuously executing so-called Chaos Experiments. A Chaos Experiment consists of two crucial elements, namely a steady-state hypothesis and an anomaly injection. Traditionally, during the experimentation process, the steady-state hypothesis is verified at the start of an experiment, after which the anomaly is injected, followed by a second evaluation of the steady-state hypothesis. This Chaos Experimentation process is well suited for identifying whether a system is in a steady state after a failure is introduced.
Problem. When applied, the traditional Chaos Experimentation approach can only verify whether the system is in a steady state without providing any information about the time between the state changes, e.g. the recovery of the system. The experimentation process does not explicitly allow the specification of hypotheses regarding the transient behavior, i.e. the behaviors experienced during the transition between steady states after a failure has been introduced. As a result, the experimentation process also does not support explicit verification of requirements on the transient behavior during the experiment, e.g, whether the response time stays below a given threshold all the time or whether a circuit breaker opens within a given time. Knowledge about such transient behaviors is beneficial for the stakeholders of the system. For example, assuming that high availability is of utmost importance for the business model of an application, a long period of recovery during which the application is unavailable after an unexpected failure could lead to considerable losses for various stakeholders.
Objective. The first objective of the thesis is to examine how a transient behavior requirement can be specified in the context of Chaos Engineering and Chaos Experiments. The next goal of the thesis is to study how the Chaos Experimentation process can include a verification of transient behavior requirements. A further goal is to create a concept for an extended Chaos Engineering approach, which supports the specification of transient behavior hypotheses and their verification, and to implement the concept into a working prototype. After the prototype is developed, the last objective of the thesis is to evaluate the extended Chaos Engineering approach and its implementation.
Method. In order to achieve the first objective of the thesis, formalisms capable of describing transient behaviors are examined with regard to their integration into Chaos Experiments. To accomplish the second objective, state-of-the-art Chaos Engineering tools are studied, and additionally, three expert interviews are conducted in order to elicit the requirements for such an extension of the Chaos Engineering approach. To accomplish the goal of creating a concept for the approach, the research results from the previous goals and the requirements elicited during the interviews are combined into a concept which is then implemented into a prototype. To reach the last goal of the thesis, the approach and the prototype are examined in three separate types of evaluation.
Result. First, the results of the thesis include the requirements for an extended Chaos Engineering approach, supporting the specification and verification of transient behavior requirements. Furthermore, the results also include a concept for the realization of the extension. Moreover, the results also comprise a functioning prototype of the proposed approach and its evaluation.
Conclusion. The proposed extension of the Chaos Engineering approach allows a deeper and more precise analysis of the resilience of software systems by enabling the specification and evaluation of more detailed and strict resilience requirements including transient behavior specifications. Furthermore, various stakeholders such as customers, operators, and developers benefit from

the extended approach, allowing them to have stricter guarantees regarding the resilience of the application. Moreover, the prototype implementing the extended approach allows software engineers to easily adopt it and possibly extend it.

Kurzfassung

Kontext. Chaos-Engineering ist ein Ansatz zur Analyse der Resilienz von Softwaresystemen, d.h. ihrer Fähigkeit, unerwarteten Ereignissen zu widerstehen, sich entsprechend anzupassen und weiterhin ihre Funktionalität bereitzustellen. Ein wesentlicher Bestandteil des Ansatzes ist das kontinuierliche Experimentieren, das sich in der kontinuierlichen Durchführung sogenannter Chaos-Experimente manifestiert. Ein Chaos-Experiment besteht aus zwei wesentlichen Elementen, nämlich einer Steady-State-Hypothese und einer Anomalie-Injektion. Traditionell wird zu Beginn eines Experiments die Steady-State-Hypothese verifiziert, danach wird die Anomalie injiziert, gefolgt von einer zweiten Bewertung der Steady-State-Hypothese. Dieses Chaos-Experimentierverfahren ist gut geeignet, um festzustellen, ob sich ein System nach dem Auftreten einer Störung in einem stabilen Zustand befindet.

Problemstellung. Bei Anwendung des traditionellen Chaos-Engineerings kann nur überprüft werden, ob sich das System in einem stabilen Zustand befindet, ohne Informationen über die Zeit zwischen den Zustandsänderungen, z. B. die Wiederherstellung des Systems, zu liefern. Der Experimentierprozess erlaubt keine explizite Spezifizierung von Hypothesen über das transiente Verhalten, d. h. das Verhalten während des Übergangs zwischen den stabilen Zuständen, nachdem ein Fehler aufgetreten ist. Infolgedessen unterstützt der Experimentierprozess auch keine explizite Verifikation von Anforderungen an das transiente Verhalten während des Experiments, z. B. ob die Antwortzeit die ganze Zeit unter einem bestimmten Wert bleibt oder ob ein Curcuit-Breaker innerhalb einer bestimmten Zeit geöffnet wird. Das Wissen über ein solches transientes Verhalten ist für die Stakeholder des Systems von Vorteil. Geht man beispielsweise davon aus, dass eine hohe Verfügbarkeit für das Geschäftsmodell einer Anwendung von größter Bedeutung ist, könnte eine lange Wiederherstellungsphase, in der die Anwendung nach einem unerwarteten Ausfall nicht verfügbar ist, zu erheblichen Verlusten für verschiedene Stakeholder führen.

Zielsetzung. Das erste Ziel dieser Arbeit ist es, zu untersuchen, wie Anforderungen an das transiente Verhalten im Rahmen von Chaos-Engineering und Chaos-Experimenten spezifiziert werden können. Das nächste Ziel der Arbeit ist es, zu untersuchen, wie der Chaos-Experimentierprozess eine Verifikation von Anforderungen an das transiente Verhalten ermöglichen kann. Ein weiteres Ziel ist es, ein Konzept für einen erweiterten Chaos-Engineering-Ansatz zu erstellen, der die Spezifikation von Anforderungen an das transiente Verhalten und deren Verifikation unterstützt, und das Konzept in einen funktionierenden Prototyp umzusetzen. Nach der Entwicklung des Prototyps ist das letzte Ziel der Arbeit die Evaluation des erweiterten Chaos-Engineering-Ansatzes und seiner Implementierung.

Methode. Um das erste Ziel der Arbeit zu erreichen, werden Formalismen, die transientes Verhalten beschreiben können, im Hinblick auf ihre Integration in Chaos-Experimente untersucht. Um das zweite Ziel zu erreichen, werden die state-of-the-art Chaos-Engineering-Tools untersucht und zusätzlich drei Experteninterviews durchgeführt, um die Anforderungen an eine solche Erweiterung des Chaos-Engineering-Ansatzes zu erheben. Zur Erreichung des Ziels, ein Konzept für den Ansatz zu erstellen, werden die Forschungsergebnisse aus den vorangegangenen Zielen und die in den Interviews erhobenen Anforderungen zu einem Konzept zusammengefasst, das anschließend in einem Prototyp umgesetzt wird. Um das letzte Ziel der Arbeit zu erreichen, werden der Ansatz und der Prototyp in drei verschiedenen Evaluationen untersucht.

Ergebnis. Die Ergebnisse der Arbeit beinhalten zum einen die Anforderungen an einen erweiterten Chaos-Engineering-Ansatz, der die Spezifikation und Verifikation von Anforderungen an das transiente Verhalten unterstützt. Des Weiteren beinhalten die Ergebnisse auch ein Konzept zur

Realisierung der Erweiterung. Weiterhin umfassen die Ergebnisse auch einen funktionierenden Prototyp des vorgeschlagenen Ansatzes und dessen Evaluation.

Fazit. Die vorgeschlagene Erweiterung des Chaos-Engineering-Ansatzes ermöglicht eine tiefere und präzisere Analyse der Resilienz von Softwaresystemen, indem sie die Spezifikation und Evaluation von detaillierteren und strengeren Resilienzanforderungen mit Anforderungen an das transiente Verhalten ermöglicht. Darüber hinaus profitieren verschiedene Stakeholder wie Kunden, Betreiber und Entwickler von dem erweiterten Ansatz, da sie strengere Garantien hinsichtlich der Resilienz der Anwendung erhalten können. Außerdem ermöglicht der Prototyp, der den erweiterten Ansatz implementiert, Software-Ingenieuren, ihn ohne weiteres zu verwenden und ihn möglicherweise zu erweitern.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ATAM Architecture Trade-Off Analysis Method. 14

CSV Comma-Separated Values. 22

CTL Computational Tree Logic. 10

DTMC Discrete-Time Markov Chain. 13

EPL Event Processing Language. 11

GQM Goal Question Metric. ix, 47

GUI Graphical User Interface. 26

HTTP Hypertext Transfer Protocol. 17

JSON JavaScript Object Notation. 8

LTL Linear Temporal Logic. 10

MTL Metric Temporal Logic. ix, 10

PCTL Probabilistic Computation Tree Logic. 13

PSP Property Specification Pattern. ix, 7, 10

QoS Quality of Service. 14

YAML YAML Ain't Markup Language. 8

1 Introduction

In the last decade, there has been a huge shift from the monolithic architectural and deployment styles to distributed and cloud-native approaches [GD18]. One reason for this shift is that the distributed and cloud-native approaches are inherently more resilient to failures, whereas monolithic applications are often prone to having single points of failure [VGC+15]. Although inherently more resilient and not subjected to single points of failure, the distributed and cloud-native applications are still not protected from failures of individual computing resources and potentially cascading failures [New15]. Additionally, the distributed nature of these applications makes the identification of such failures and the assessment of their magnitude more complex [New15]. One approach, aiming to support software engineers in analyzing and preventing such failures, is Chaos Engineering.

The Chaos Engineering approach assesses the resilience of software systems by deliberately injecting failures or anomalies as part of experiments. The so-called Chaos Experiments are combining the injection of anomalies with validations of hypotheses regarding a steady state of the system's operation before and after the insertion of the anomaly. Experimentation is also a defining property of the approach, as experimentation creates new knowledge compared to testing which only verifies existing knowledge [RJ20]. Furthermore, Chaos Engineering should be applied iteratively and continuously, so that it can assess the resilience of the software after changes have been introduced and at various points during the software development life cycle [RJ20].

The anomalies injected as part of Chaos Engineering cause the target systems to experience transient behaviors. The term transient behavior is originating from the field of electrical engineering and is referring to the change of states when a system is transitioning from one steady state to another [VFH14]. By deliberately injecting anomalies such as stopping services, introducing sudden workload changes, and triggering auto-scaling policies, the Chaos Experiments are triggering the resilience mechanisms of the target systems and are causing them to change states and experience transient behaviors. The state-changing behaviors of adaptive systems have already been studied and PSPs [DAC99] and temporal logic [AGL+15] have been identified as capable of formally specifying such behaviors [CL12]. Even though these behaviors are already known and studied, the behaviors and their verification have not been studied with regard to Chaos Engineering and in particular the Chaos Experimentation process.

Problem Statement

The Chaos Experimentation process, in particular the verification of the steady-state hypothesis before and after the introduction of anomalies, is capable of providing insight into the resilience of an application and whether it is able to adapt to failures and continue operating properly after such are introduced. However, this process is conceptually limited as the Chaos Experiments only allow the specification of steady-state hypotheses, meaning that a verification of the transient behaviors exhibited during state changes is not explicitly considered. Even though different Chaos Engineering tools offer the possibility to configure the timing of the steady-state hypothesis checks and even to carry out such continuously during the whole experiment, the Chaos Experiments are still limited since the steady-state hypotheses are meant to make statements only regarding the stable state of the system. Additionally, the transient states might have different characteristics and acceptance thresholds. To better illustrate this, consider an example, in which there is a system consisting of multiple services, a steady-state hypothesis that the response time of the system is under 1 second, and a failure injection which suddenly stops one of the services. It is also assumed that the response time of the system is an important quality for the users of the system and a sudden increase in it could lead to the loss of users which is also a loss for various stakeholders of the application. Therefore, the stakeholders of the application require guarantees regarding the transient phases in which they could lose users. As a result, the stakeholders define the transient behavior requirement that after a failure of a service, the response time of the system could increase to 2 seconds for a time period not exceeding 30 seconds, and the stopped instance should be restarted during these 30 seconds. By using the traditional Chaos Engineering approach, first, it would not be possible to define the transient behavior requirement as part of the Chaos Experiment using the supported constructs, and second, a verification of the transient behavior requirement would not be possible even if the steady-state hypothesis check is performed continuously. On the contrary, the extended Chaos Engineering approach proposed in this thesis allows the specification of transient behavior hypotheses and their verification.

As modern software systems are often consisting of numerous highly distributed components, failures or anomalies of single components are not uncommon. As a result, the system spends a certain time transitioning between states, i.e. experiencing transient behaviors. Additionally, such behaviors and state changes can be triggered by various stimuli. Examples of such are rolling deployments when both old and new versions of a system are running simultaneously, failures of components when requests are rerouted to stable components and the failed components are restarted, and workload changes when resources have to be dynamically added or removed [Bec20]. Since these behaviors can have an influence on the overall usage of the system, a closer look into the transient behaviors is beneficial to the stakeholders of the software. Another problem is that a formal method for specifying transient behaviors in Chaos Engineering has not been researched yet. Formal methods such as temporal logic have been already studied with regards to specifying temporal behaviors, however, the applicability of these methods to specifying transient behaviors as part of Chaos Experiments and an approach for integrating them into Chaos Engineering have not been studied yet. Furthermore, introducing support for the analysis of formal specifications of such behaviors as part of the Chaos Experimentation process would allow a deeper insight into the resilience of software applications and also enable the specification and verification of more fine-grained resilience requirements.

Objective

The first objective of the thesis is to study how a transient behavior requirement specified using a formalism capable of describing transient behaviors and their temporal properties can be incorporated into Chaos Experiments. The results regarding this objective will also provide the answers to the first research question of the thesis which is the following:

RQ1 How can a transient behavior requirement be specified in the context of Chaos Engineering and Chaos Experiments?

The second objective of this thesis is to investigate how the Chaos Experimentation process can be extended to support the verification of transient behavior requirements. In addition, this objective also includes eliciting requirements regarding such an extension of the Chaos Experimentation process from individuals with Chaos Engineering expertise. The findings regarding this objective will respectively answer the following research question:

RQ2 How can the Chaos Experimentation process incorporate a verification of transient behavior requirement specifications?

After defining how the Chaos Engineering approach can be extended and gathering requirements for this extension, a further objective of the thesis is to create a final concept for the proposed extension and to develop a prototype with the goal of evaluating the proposed extension. This is reflected in the following research question:

RQ3 What does a concept for an extended Chaos Engineering approach contain and how can it be implemented into a prototype?

A further goal of the thesis is to evaluate the proposed extended Chaos Engineering approach and its implementation with regard to different criteria, namely the correctness of the verification of the transient behavior specifications, the execution of Chaos Experiments, and the compatibility with other approaches. The results of the evaluation will provide answers to the following research questions:

RQ4 Is the extended Chaos Engineering approach able to correctly verify different transient behavior requirements?

RQ5 Is the extended Chaos Engineering approach capable of correctly executing Chaos Experiments and verifying transient behavior requirements as part of the experiments?

RQ6 Is the extended Chaos Engineering approach compatible with other approaches and tools?

Method

To achieve the first goal of the thesis, the temporal logic formalisms that have already been identified as capable of describing transient behaviors are studied with regard to how they can be incorporated as transient behavior requirements into Chaos Experiments. To achieve the second goal of finding out how the Chaos Experiments can support the verification of transient behavior requirements, state-of-the-art Chaos Engineering tools are examined. Additionally, the requirements for such an extension of the Chaos Experimentation process are gathered in three interviews with people with Chaos Engineering experience. After completing the interviews, in order to accomplish the third goal of the thesis, the requirements elicited during them and the information gathered during the research regarding the previous objectives are used to develop a final concept for the extended Chaos Engineering approach. After the concept is created, it is implemented into a prototype. Once the implementation of the prototype is completed, it is evaluated with regard to various criteria. First, the prototype is evaluated with regard to the correctness of the verification of transient behavior specifications. For this, a large variety of transient behavior specifications is verified using the prototype in order to find out how many of these get correctly verified. Afterward, the ability to carry out this verification as part of the Chaos Experimentation process is evaluated by executing a variety of Chaos Experiments including transient behavior hypotheses on a test system. Furthermore, the compatibility of the proposed approach with other tools and approaches is studied by using the verification of transient behavior specification on the data produced by a tool capable of simulating microservice architectures and their resilience.

Results

First, the extension of the Chaos Experiments with a transient behavior specification using formalisms such as temporal logic and PSPs, and the extension of the Chaos Experimentation process by the inclusion of runtime verification are studied. The resulting initial concepts from this research are combined into an initial prototype which is used during the three expert interviews that successfully elicited requirements for the extended Chaos Engineering approach. After these are taken into consideration, a final concept for the extension of the Chaos Engineering approach, which among others includes a transient behavior hypothesis section in the Chaos Experiment and an additional transient behavior verification step in the experiment execution is created. The concept is then successfully implemented into a prototype with the aim to evaluate the proposed extension. All of the planned evaluations are successfully carried out and their results prove that the extension of the Chaos Engineering approach achieves its purpose and is a viable extension to the traditional Chaos Engineering approach.

Thesis Structure

In the following, the chapters of this thesis proposal are listed and their content is described.

Chapter 2 – Foundations and Related Work: This chapter provides valuable information about the theoretical foundations, the technical background of the thesis, and related research in this area.

Chapter 3 – Elicitation of Requirements for the Extended Chaos Engineering Approach: This chapter presents the interview process, initial concepts presented to the interviewees, the interview questions, and the results of the interviews.

Chapter 4 – Extended Chaos Engineering Approach: In this chapter, the concept for the extended Chaos Engineering approach is defined and thoroughly described.

Chapter 5 – Prototype Implementation: In this chapter, the technical details regarding the implementation of the concept into a working prototype are discussed.

Chapter 6 – Evaluation: In this chapter, the extended Chaos Engineering approach and its implementation are evaluated with regard to the correctness of the verification of transient behavior specifications, the integration of the verification into the Chaos Experimentation process, and the compatibility with other approaches.

Chapter 7 – Conclusion and Future Work: This chapter concludes the thesis and discusses possible future work extending this thesis.

Furthermore, all supplementary materials such as the source code of the prototype and all evaluation resources are publicly available [Zah22].

2 Foundations and Related Work

This chapter discusses the foundations of this thesis and related research.

2.1 Foundations

In the following sections, the theoretical foundations of this work, such as Chaos Engineering, temporal logic, Property Specification Patterns (PSPs), and runtime verification, as well as the technical background required for a proper understanding of the work are described in detail.

2.1.1 Chaos Engineering

Chaos Engineering is an approach pioneered by Netflix, in which failures are deliberately caused while the system is operating in a normal production environment, in order to observe if the system is resilient to such failures [RJ20]. The Chaos Team at Netflix also came up with a definition and a blueprint of the Chaos Engineering approach called "The Principles" [Pri] in which the approach is formally defined as "*the discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production*" [RJ20]. Additionally, the five steps of the Chaos Experimentation process are defined as follows: (1) create a hypothesis regarding the steady-state behavior of the system, i.e. a steady-state hypothesis, (2) select experiment variables reflecting real-world events, (3) run experiments in production, (4) automate experiment execution, and (5) minimize the blast radius [RJ20]. As mentioned, two of the elements of a Chaos Experiment are the steady-state hypothesis and the injection of an anomaly or a failure. The steady-state hypothesis defines a statement regarding the steady state of the application, i.e. the state of its normal and stable operation. One example of a steady-state hypothesis could be the following statement: "The `/getUsers` URL responds with a `200` status code, within 1 second" [Mil19]. An example of a failure that could be defined as part of a Chaos Experiment could be the following: "Disconnect the database cluster from the network" [Mil19]. The execution of a Chaos Experiment often starts with a verification of the steady-state hypothesis before the failure is injected. This is done in order to verify that the system is stable before any disruptions are introduced. If this holds, the anomaly is introduced, i.e. the failure is injected. Afterward, the steady-state hypothesis is verified once again, in order to examine if the system is behaving normally after the failure was introduced.

2.1.2 Chaos Toolkit

Chaos Toolkit [Chac] is a tool that enables the execution of automated Chaos Experiments [Mil19]. Its main advantages are that it is open-source and has a variety of extensions that allow the injections of faults in multiple layers, namely platform, network, and application [Chac]. Chaos Toolkit supports Chaos Experiments specified as either JavaScript Object Notation (JSON) or YAML Ain't Markup Language (YAML) files.

Chaos Toolkit's Chaos Experiments usually contain two components, a steady-state hypothesis, and a fault injection, also called method. The steady-state hypothesis usually contains probes, which are calls retrieving information from the system under test, while the method usually contains actions, which respectively perform operations on the system under test [Chac]. In Listing 2.1 an example experiment specified using Chaos Toolkit's syntax is presented. In this example, the experiment contains a simple steady-state hypothesis, namely that the microservices of the system remain healthy, and a simple fault injection, which is expressed in killing a random microservice instance. Additionally, the probe in the steady-state hypothesis and the action in the method are making use of the Kubernetes extension for Chaos Toolkit, which integrates various Kubernetes operations into Chaos Toolkit's probes and actions [Chac].

2.1.3 Transient Behaviors

The terms transient behavior and transient state originate from the field of electrical engineering. Venikov [VFH14] defines transient states as "occurring when the system is changing from one steady state to another". Similarly to electrical systems, software systems are also able to transition from one steady state into another while experiencing certain behaviors during this transition. As a result, the terms transient behavior and transient state can also be applied to the behaviors occurring in software systems. These terms can also be particularly useful in approaches like Chaos Engineering that are examining the steady states and their changes when anomalies are introduced.

The causes for transient behavior in software systems can vary widely. Some possible causes for such behaviors have been discussed in the work of Beck [Bec20] and are presented in this paragraph. One of the possible causes can be rolling deployments, during which both the old and the new version of the software are simultaneously running, and as a result, the system is in an undefined state of transient behavior during the deployment. Another cause can be the failure of components and the triggering of resilience mechanisms, e.g. when the requests to a failed instance are rerouted to another one while the failed one gets restarted. During this time the system would again be exhibiting transient behaviors. A further cause could be sudden changes in user behavior and respectively in the workload, as a result of which the autoscaling of the application might initiate the process of starting or stopping instances, during which the system would be experiencing transient behaviors.

2.1.4 Temporal Logic

The term temporal logic covers a variety of logical formalisms that enhance the traditional propositional logic with temporal modalities to specify and reason about timing properties of logical expressions [AH92]. Two of the most common temporal modalities present in most temporal

Listing 2.1 Chaos Toolkit Example Experiment [Chac]

```
{
    "title": "Do we remain available in face of pod going down?",
    "description": "We expect Kubernetes to handle the situation gracefully when a pod goes
↪ down",
    "tags": ["kubernetes"],
    "steady-state-hypothesis": {
        "title": "Verifying service remains healthy",
        "probes": [
            {
                "name": "all-our-microservices-should-be-healthy",
                "type": "probe",
                "tolerance": true,
                "provider": {
                    "type": "python",
                    "module": "chaosk8s.probes",
                    "func": "microservice_available_and_healthy",
                    "arguments": {
                        "name": "myapp"
                    }
                }
            }
        ]
    },
    "method": [
        {
            "type": "action",
            "name": "terminate-db-pod",
            "provider": {
                "type": "python",
                "module": "chaosk8s.pod.actions",
                "func": "terminate_pods",
                "arguments": {
                    "label_selector": "app=my-app",
                    "name_pattern": "my-app-[0-9]$",
                    "rand": true
                }
            },
            "pauses": {
                "after": 5
            }
        }
    ]
}
```

logics are the following operators: $\Diamond$ - eventually (eventually in the future), $\Box$ - always (now and forever in the future), and U - until [BK08]. Some of these temporal logics can also be extended with past operators such as the following: $\Diamond^{-1}$ - sometime in the past, and $\Box^{-1}$ - always in the past [BK08]. There are two main types of temporal logics, namely linear-time and branching-time logics [AH92]. In linear-time logics, the logical statements are interpreted over linear structures of states, meaning that there is a single linear trace of states. On the contrary, in branching-time logics, the statements are interpreted over tree structures of states, where there are multiple different paths, any one of which might be realized [AH92]. One example of a temporal logic based on linear time is the Linear Temporal Logic (LTL). An example logical expression for the response property "*every request will eventually lead to a response*" specified using LTL is the following: $\Box(request \rightarrow \Diamond response)$ [BK08]. Another example, this time for a temporal logic based on branching time, is Computational Tree Logic (CTL) [BK08]. The response property introduced above specified using CTL is the following: $\forall\Box(request \rightarrow \forall\Diamond response)$ [BK08]. Even though the two examples might look similar, as LTL and CTL share a lot of logical operators, the expressiveness of LTL and CTL is incomparable [BK08].

2.1.5 Metric Temporal Logic

Metric Temporal Logic (MTL) is an extension of LTL in which the temporal operators are time-bounded by the introduction of timing constraints in the form of time intervals [Ulu19a]. Additionally, MTL supports past and future operators. As a result, MTL is often split semantically into two subvariants, namely past-MTL and future-MTL, depending on the utilized temporal operators. In the following, only the syntax of past-MTL and examples regarding this variant are presented, since this is the MTL variant that is primarily used in the remainder of the thesis.

Past-MTL uses the following temporal operators: *past eventually* ($\blacklozenge_I$), *past always* ($\blacksquare_I$), and *since* (S_I) [Ulu19b]. Given a finite set P of atomic propositions, past-MTL formulas are defined by the following grammar [Ulu19b]:

$$\varphi = \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \, \mathcal{S}_I \, \varphi_2$$

where $p \in P$ and $I \subseteq [0, \infty)$. In the above-presented notation, the subscript I at each temporal operator represents the time interval, i.e. the timing constraint of the given operator. For example, the MTL formula $\blacklozenge_{[0,5]}Q$ means that Q will hold eventually within 5 time units. The satisfaction relation $(w, t) \vdash \varphi$ indicates that the temporal behavior $w : \mathbb{T} \rightarrow \mathbb{B}^P$ satisfies the formula φ at time point $t \in \mathbb{T}$ as follows [Ulu19b]:

$$(w, t) \vdash p \Leftrightarrow w_p(t) = \top$$

$$(w, t) \vdash \neg\varphi \Leftrightarrow (w, t) \nvdash \varphi$$

$$(w, t) \vdash \varphi_1 \vee \varphi_2 \Leftrightarrow (w, t) \vdash \varphi_1 \text{ or } (w, t) \vdash \varphi_2$$

$$(w, t) \vdash \varphi_1 \mathcal{S}_I \varphi_2 \Leftrightarrow \exists t' \in (0, t).\ (w, t') \vdash \varphi_2 \text{ and } \forall t'' \in (t', t).\ (w, t'') \vdash \varphi_1 \text{ and } t - t' \in I$$

2.1.6 Property Specification Patterns

Property Specification Patterns (PSPs), introduced by Dwyer et al. [DAC99], are a system of recurring solutions, dealing with the temporal intricacies of reactive systems [AGL+15]. These patterns provide general rules that help practitioners to qualify order and occurrence, quantify time

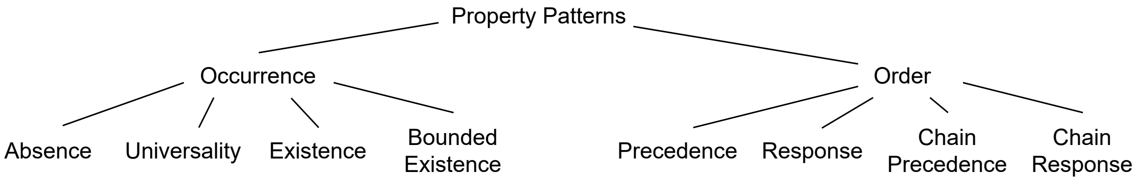


Figure 2.1: Property Specification Patterns - Original Hierarchy [DAC99]

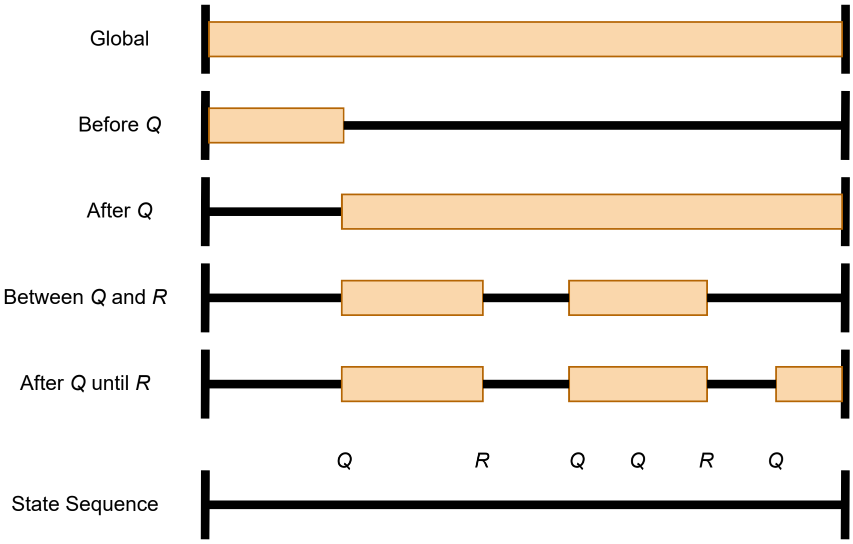


Figure 2.2: Property Specification Patterns - Scopes [DAC99]

bounds, and express probabilities of events [AGL+15]. Figure 2.1 depicts an overview of the pattern hierarchy. Each of the patterns describes a commonly occurring requirement on the event sequences in a finite-state system model [DAC99]. In the following, three example patterns are listed, the Existence and Bounded Existence patterns from the Occurrence category and the Response pattern from the Order category:

Existence A given event must occur within a scope.

Bounded Existence A given event must occur k times within a scope

Response A given event P must always be followed by an event S within a scope.

As can be seen in the three example patterns listed above, the patterns also contain a scope. Dwyer et al. [DAC99] define the scopes presented in Figure 2.2.

Additionally, for each pattern Dwyer et al [DAC99] specify mappings for the pattern into the previously introduced temporal logics, namely LTL and CTL. A paper by Czepa and Zdun [CZ18] compared the PSPs to LTL and Event Processing Language (EPL) with regard to their understandability. The authors found evidence supporting their hypothesis that PSPs are more easily understandable than the rest of the studied formalisms [CZ18]. Throughout the years, additional patterns have been introduced into the pattern catalog and new mappings into additional temporal logic formalisms have been introduced [AGL+15]. The new extended version of the PSP catalog is

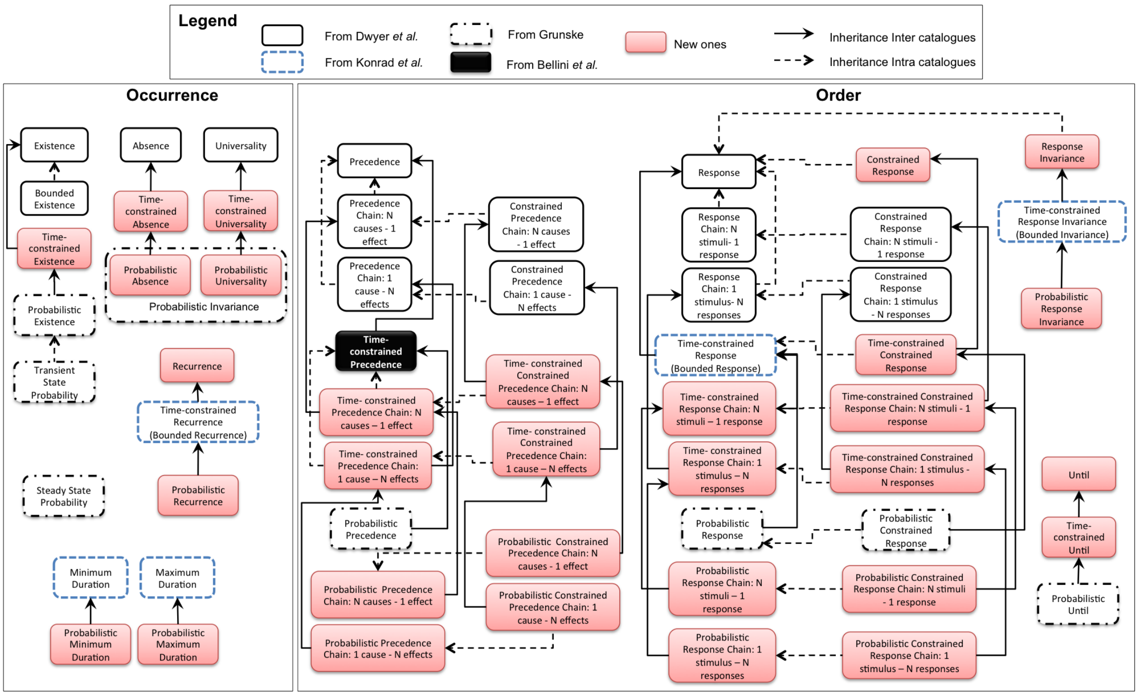

Legend
From Dwyer et al.
From Konrad et al.
From Grunske
From Bellini et al.
New ones
Inheritance Inter catalogues
Inheritance Intra catalogues
Occurrence
Existence
Bounded Existence
Time-constrained Existence
Probabilistic Existence
Transient State Probability
Absence
Time-constrained Absence
Universality
Time-constrained Universality
Probabilistic Absence
Probabilistic Universality
Probabilistic Invariance
Recurrence
Time-constrained Recurrence (Bounded Recurrence)
Probabilistic Recurrence
Steady State Probability
Minimum Duration
Maximum Duration
Probabilistic Minimum Duration
Probabilistic Maximum Duration
Order
Precedence
Precedence Chain: N causes - 1 effect
Precedence Chain: 1 cause - N effects
Constrained Precedence Chain: N causes - 1 effect
Constrained Precedence Chain: 1 cause - N effects
Time-constrained Precedence
Time- constrained Precedence Chain: N causes – 1 effect
Time- constrained Precedence Chain: 1 cause – N effects
Time- constrained Constrained Precedence Chain: N causes – 1 effect
Time- constrained Constrained Precedence Chain: 1 cause – N effects
Probabilistic Precedence
Probabilistic Constrained Precedence Chain: N causes - 1 effect
Probabilistic Precedence Chain: N causes - 1 effect
Probabilistic Constrained Precedence Chain: 1 cause - N effects
Probabilistic Precedence Chain: 1 cause - N effects
Response
Response Chain: N stimuli- 1 response
Response Chain: 1 stimulus- N responses
Constrained Response
Constrained Response Chain: N stimuli - 1 response
Constrained Response Chain: 1 stimulus - N responses
Time-constrained Response (Bounded Response)
Time-constrained Constrained Response
Time- constrained Response Chain: N stimuli – 1 response
Time-constrained Constrained Response Chain: N stimuli - 1 response
Time- constrained Response Chain: 1 stimulus – N responses
Time-constrained Constrained Response Chain: 1 stimulus - N responses
Probabilistic Response
Probabilistic Constrained Response
Probabilistic Response Chain: N stimuli – 1 response
Probabilistic Constrained Response Chain: N stimuli - 1 response
Probabilistic Response Chain: 1 stimulus – N responses
Probabilistic Constrained Response Chain: 1 stimulus – N responses
Response Invariance
Time-constrained Response Invariance (Bounded Invariance)
Probabilistic Response Invariance
Until
Time-constrained Until
Probabilistic Until

Figure 2.3: Property Specification Patterns - Extended Catalog [AGL+15].

presented in Figure 2.3. Furthermore, a study by Grunske et al. [AGL+15] introduced a software tool called PSP Wizard, allowing its users to easily create PSP specifications in natural language and transform these into expressions in various temporal logic formalisms.

2.1.7 Propositional Functions and Predicates

Throughout the thesis, the terms predicate and predicate function are used, thus they must be defined. A predicate or also called a propositional function is an expression that contains a variable and becomes a statement when a constant is substituted into the variable [CCR13]. A predicate normally has $true$ and $false$ substitution instances [CCR13]. For example, when P represents a human, s represents Socrates, and c represents Chicago, then $P(s)$ is $true$ while $P(c)$ is $false$ [CCR13]. In the remainder of the thesis, the term predicate is used instead of the term propositional function. Furthermore, predicates are used in MTL formulas and PSPs similarly to how propositions are used.

2.1.8 Runtime Verification

Leucker and Schallhart [LS09] define runtime verification as a discipline dealing with the techniques enabling checking whether a run of a system satisfies or violates a given correctness property. Leucker and Schallhart also define a component called a monitor, which is responsible for reading a finite event trace and delivering a certain yes/no or $true/false$ verdict. Even though runtime verification has its origins in model checking, the two approaches are inherently different according to the authors. Model-checking typically examines all executions of a system in order to answer whether it satisfies a given property. On the contrary, runtime verification considers only a single execution. Additionally, model checking usually considers infinite traces while runtime verification considers finite ones [LS09]. Runtime verification has also been studied with regards to verifying properties specified using temporal logic formalisms such as LTL and MTL, and approaches to achieve this verification and to build the required monitors have already been proposed [BLS07; Ulu19b].

Ulus [Ulu19b] proposes an approach for creating monitors for the past variant of MTL using sequential networks. The sequential networks used in the approach are abstract machines with finite state spaces, a set of update rules, and an output function [Ulu19b]. Even though sequential networks are functionally identical to finite automata, they differ in their structure [Ulu19b]. These structural differences make the sequential networks particularly suitable to create runtime verification monitors [Ulu19b]. The approach proposed by Ulus considers only the past fragment of MTL because past-oriented monitoring is inherently less costly than future-oriented monitoring and because otherwise, the monitor would have to be able to delay the output or output $uknown$ [Ulu19b]. Furthermore, Ulus presents an implementation of his approach and compares it with other tools providing past-MTL monitors. The results of this comparison indicate that Ulus' approach handles large time constraints over long sequences more efficiently than the other monitoring tools [Ulu19b].

2.2 Related work

In the following sections, related research regarding the specification and verification of transient behaviors using the already introduced formalisms, an extension of the Chaos Engineering approach, and an approach to visualize transient behaviors are discussed.

2.2.1 Specifying and Verifying Transient Behaviors Using Temporal Logic

Camara and De Lemos [CL12] propose an approach for verification of self-adaptive systems and resilience properties based on stimulation and probabilistic model-checking [CL12]. The approach of the authors is to stimulate the system's environment to exercise its adaptive capabilities and to capture data about the system's reaction to the stimulus [CL12]. The collected data is then transformed into Discrete-Time Markov Chains (DTMCs). The authors also specify the resilience properties they want to verify as Probabilistic Computation Tree Logic (PCTL) expressions using the response pattern from the PSPs. The researchers then verified the PCTL expressions against the DTMCs using model-checking. The logical language chosen by the researchers is a variant of the CTL language, which is suitable for expressing temporal properties, typical for transient behaviors. The DTMC models, however, are not representative of the techniques utilized by Chaos Engineering, since DTMCs are a probabilistic approach, used to generalize the results of multiple executions. The aim of this thesis is to enable exact statements regarding transient behaviors in a single experiment run. This thesis also differs from the work of Camara and De Lamos [CL12] as it focuses on the verification of transient behaviors as part of the Chaos Experimentation process, excluding the assumptions and the probabilistic properties of the DTMC approach.

2.2.2 Extending the Chaos Engineering Approach

Frank et al. [FHW+21] conducted a study with an industry partner, the aim of which was to explore the application and adoption of the scenario-based Architecture Trade-Off Analysis Method (ATAM) for the elicitation of resilience requirements. The authors conducted an ATAM-based workshop with the industry partners and collected resilience scenarios which were then turned into Chaos Experiments. The Chaos Experiments were executed on a mock system provided by the industry partner and the authors were able to identify weaknesses. The above-mentioned goal and employed methods are not particularly concerning this thesis, however, the experimental setup used by the researchers is of interest for this thesis. The authors used Chaos Toolkit [Chac], which is a tool allowing software engineers to perform Chaos Experiments, together with a load generator, generating a steady amount of requests to the system under test during the whole experiment. The response times of the requests were also captured and saved in an InfluxDB database. Additionally, the authors also implemented their own hypothesis validator. This experimental setup allowed the researchers to measure detailed information about the system under test's response times during the whole duration of the Chaos Experiments, i.e. the approach could capture the transient behaviors regarding the response time. Even though the approach was able to capture these transient behaviors, it did not support a verification of the experienced behaviors against a given transient behavior requirement. This is where this thesis differs from the work of Frank et al. since the Chaos Engineering approach that it proposes includes the specification of transient behavior requirements as part of the Chaos Experiments and provides explicit verification of the specified requirements.

2.2.3 Visualizing Transient Behaviors

Frank et al. [BFH+21] conducted another study, as part of which the authors present TransVis, an approach to support software architects and DevOps engineers to visually specify and assess transient behaviors in microservice systems [BFH+21]. The approach includes a dashboard, integrating visualizations of the microservice architecture, the transient behaviors requirement specifications, and the actually experienced transient behaviors. The researchers also integrated a chatbot into their approach, enabling interactions with the visualizations through natural language. After introducing TransVis, the authors conducted an exploratory expert study aiming to examine how well the approach supports software architects and DevOps engineers in the specification and verification of transient behaviors [BFH+21]. The results of the study indicated that the developed visualizations are effective, especially for exploring transient behavior and comparing a service's Quality of Service (QoS) with a specified behavior, however, the chatbot did not prove to be significantly beneficial [BFH+21]. Even though the study by Frank et al. [BFH+21] is not concerning the area of Chaos Engineering, it proposes a way to specify transient behavior using a formalism, namely a resilience triangle [BCE+03] consisting of three fixed metrics. This is also where the approach for specifying transient behaviors of Frank et al. differs from the one proposed by this thesis, which utilizes temporal logic and PSPs. Additionally, the approach proposed by Frank et al. allows only a manual verification in the form of comparisons of the transient behavior specifications and the actual behaviors which is where this thesis also differs from the work of Frank et al. as the approach proposed in this thesis automates this verification as part of the Chaos Experimentation process.

3 Elicitation of Requirements for the Extended Chaos Engineering Approach

In addition to identifying a means to specify transient behavior requirements as part of Chaos Experiments and how such requirements can be verified during the execution of the experiments, a further goal of the thesis is to gather requirements for an extended Chaos Engineering approach incorporating these specifications and their verification. For this purpose, a number of interviews were conducted with people that are practicing the Chaos Engineering techniques in industry and research.

3.1 Goals

The first goal of the interviews was to explore whether people that have experience with Chaos Engineering have already tried to examine transient behaviors using Chaos Engineering and have experienced difficulties while specifying the behaviors that they wanted to verify. Another goal of the interviews was to find out whether the interviewees agree with the concepts and ideas that can be used to realize the extended Chaos Engineering approach. The last goal of the interviews was to elicit the requirements of the interviewees regarding the extended Chaos Engineering approach, including requirements on the conceptual level as well as the technical realization of the implementation.

3.2 Method

The candidates that were invited to participate in an interview were either people involved in the implementation of state-of-the-art Chaos Engineering tools, people working in the industry and practicing Chaos Engineering techniques, or people involved in Chaos Engineering research. When the candidates were contacted they also received an invitation document containing a short description of the interview and a consent form. In total 7 people were contacted of which 4 answered with interest in participating, however, one of them was not able to schedule an interview. As a result, there were three participants and respectively three interviews in total. As arranged with the interviewees, all collected data was kept confidential, so any personal information that could reveal the identity of the interviewees has been obscured in the remainder of the thesis.

The first interviewee is a senior software architect with a doctorate degree in computer science. The second interviewee is a master's student who has done research in the domain of Chaos Engineering and is working as a student research assistant at the time of writing. The third interviewee is also a master's student involved in Chaos Engineering research, working and applying Chaos Engineering in the industry.

The interviews were conducted online as video calls and in a semi-structured manner. Each interview lasted between 60 and 90 minutes. Additionally, the interviews were recorded with the participant's consent. The interviews consisted of three parts which were structured as follows:

1. **Introduction and Motivation:** In this part, the interviewees were given a short introduction and were asked a series of introductory questions with the goal of finding out whether they have experienced the problems that the approach proposed in this thesis aims to solve.
2. **Initial Ideas and Concepts:** In this part, the interviewees were presented with some initial ideas and concepts regarding the extended Chaos Engineering approach. Furthermore, they were also presented with some technologies and approaches that could be used to implement the concepts into an initial prototype. In addition, this part included a video demonstration of the initial prototype.
3. **Requirements Elicitation:** In this part, the interviewees' requirements regarding the extended Chaos Engineering approach were elicited with the help of a predefined question catalog. The question catalog was split into three parts (excluding the introductory questions in the first part of the interview), namely conceptual, technical, and non-functional requirements. The conceptual part contained questions regarding the theoretical concepts in the extended Chaos Engineering approach. The technical part consisted of questions regarding the technical realization of the concept. The non-functional requirements part included questions targeting aspects like usability, performance, and extensibility.

In the following sections, the contents of the individual parts of the interview are described. Since the introduction to the thesis and the motivation behind it were already covered in the previous chapters, the following sections focus mainly on the initial ideas and concepts that were part of the initial prototype shown to the interviewees and the requirements elicitation questions that the interviewees were asked.

3.2.1 Initial Concepts and Prototype

The research regarding the first two goals of the thesis, namely to identify how transient behavior requirements can be specified as part of Chaos Experiments and how a verification of such requirements can be carried out during the execution of Chaos Experiments, resulted in the gathering of initial concepts and ideas for the realization of the extended Chaos Engineering approach. These ideas were combined into an initial prototype with the goal of finding out whether the ideas are suitable for the creation of the final concept of the extended Chaos Engineering approach and its implementation. In the sections below, the initial ideas used to construct the initial prototype presented in the interviews are described.

Transient Behavior Specification Formalism

As already mentioned in Chapter 2, there are two types of temporal logics, namely linear-time logics, which are interpreted over a single linear trace of events, and branching-time logics, which are interpreted over tree structures of states with multiple possible paths. To identify which interpretation of time is better suited for the needs of Chaos Engineering, the properties of the Chaos Engineering approach itself have to be considered. Each Chaos Experiment that gets executed is producing a single execution run of the system under test. In these runs, there are no multiple possible states in different branches and the observed state traces can be considered linear. As a result, the branching-time logics can be discarded because of their tree-like state interpretation and linear-time logics are the more suitable type of temporal logics. Two prominent examples of linear-time temporal logics have already been introduced in Chapter 2, namely LTL and MTL. Since MTL is an extension of LTL that adds temporal constraints to the temporal operators of LTL, which are particularly useful for specifying requirements concerning the transient phases of the Chaos Experiments while maintaining the original syntax of LTL, MTL proves to be the most viable temporal logic formalism for the proposed extension of the chaos engineering approach.

Moreover, the PSPs that were also introduced in Chapter 2 are another promising formalism for expressing transient behavior specifications, especially considering the fact that the users of the extended Chaos Engineering approach might not have prior experience with temporal logic. Therefore, a particularly useful property of the PSPs is that there are already existing mappings of the patterns into temporal logic formalisms including MTL. This additionally removes the need to learn how to express complex behaviors with the logical syntax of MTL and allows an easier definition using only the patterns and then transforming the patterns into MTL expressions. As a result, the PSPs are also considered for the extended Chaos Engineering approach as an abstraction layer for the complexity of MTL.

Transient Behavior Verification

As already mentioned in Chapter 2, there are already existing approaches and implementations of runtime verification monitors for MTL expressions. Nevertheless, the integration of such monitors with the existing Chaos Engineering tools has to be considered. Namely, a method for triggering the runtime verification and exchanging information between the Chaos Engineering tools and the runtime verification monitor has to be identified. One way to do this is by utilizing the Hypertext Transfer Protocol (HTTP) protocol which is supported by most Chaos Engineering tools as it is used to perform various actions during the execution of Chaos Experiments.

Transient Behavior Monitoring

The approach of runtime verification requires an event trace of the occurring behaviors in order to be able to deliver verification results. Therefore, an extended Chaos Engineering approach requires the introduction of an additional component, the purpose of which is the monitoring of the events and the creation of the event traces.

The Initial Prototype

Combining the ideas presented above results in the creation of an initial prototype, used during the interviews with the experts in order to present them with the initial conceptual ideas and demonstrate how such an approach would function. In this initial prototype, the transient behavior specifications are defined as JSON objects which include an MTL formula, information regarding the logical variables in the formula, and information regarding the location and the retrieval of the event traces required for the verification of the specification. Regarding the verification of these specifications, the initial prototype uses the already existing approach by Ulus [Ulu19b] and its implementation as a Python library [Ulua]. Additionally, the functionalities of the library are provided by a REST API in order to easily enable communication over the HTTP protocol. For the monitoring of the occurring behaviors and the creation of the event traces, a proprietary load generator and an InfluxDB database are used. These are selected since they are already a part of a test system provided by the supervisors of this thesis for the purpose of executing Chaos Experiments on it [FHW+21]. Furthermore, this test system is also used to demonstrate the capabilities of the initial prototype during the expert interviews.

3.2.2 Hypotheses

In this section, several hypotheses are stated that should be either confirmed or rejected after the interviews are conducted and the interviewees' answers are collected. The purpose of the hypotheses is to assess whether the respondents agree with the problems that the extended Chaos Engineering approach is aiming to solve and the concepts that will be included in the extended approach. The results regarding each hypothesis and whether the hypotheses are confirmed or rejected will affect the final concept of the extended approach by determining what will be included in it.

H_1: The interviewees have experienced situations in which they were not able to define the behaviors that they wanted to verify.

H_2: The interviewees have tried to analyze behaviors that were not regarding the steady state and they were unable to achieve this.

H_3: The interviewees think that the proposed extension to the Chaos Engineering approach will provide a deeper insight into the resilience of software systems and view it as a meaningful extension to the traditional Chaos Engineering approach.

H_4: The interviewees would like to be able to use the verification of transient behavior specifications outside the scope of Chaos Engineering, i.e., as a stand-alone analysis.

H_5: The concept regarding the specification of transient behavior requirements using MTL and PSPs is sufficient and complete.

H_6: The concept regarding the verification of transient behavior requirements using runtime verification monitors is sufficient and complete.

H_7: A visualization containing the verification results and the monitoring data is a meaningful output of the approach.

H_8: Usability, extensibility, and performance are important non-functional requirements for the proposed approach and its prototype.

3.2.3 Requirements Elicitation Questions

There were four question categories in total, which are listed below. For each category, a short description of the questions it contains and the motivation behind them is provided.

Introductory questions:

The questions of this category are asked after the first part of the interview, namely after the presentation of the introduction and the motivation. The purpose of these questions is to learn more about the expertise of the interviewees, find out if they have already tried to analyze transient behaviors using Chaos Engineering and whether they have successfully achieved this.

1. How familiar are you with Chaos Engineering and how would you rate your expertise with it?
2. Do you agree with the presented problem statement?
   - 2.1. Have you ever experienced situations in which you were not able to define the behaviors that you wanted to verify?
   - 2.2. Have you ever experienced situations in which you wanted to analyze behaviors that were not regarding the steady state and did you successfully achieve this? For example, knowing that a system would deviate from the steady state and analyzing how it actually deviates.

Conceptual questions:

The questions in this category are asked with the intention of finding out what the interviewees think about the concepts, such as MTL, the PSPs, and runtime verification, which are presented to them, how they would use the proposed approach, and whether they have any suggestions or recommendations regarding the presented concepts. This category contains the following questions:

3. Do you think that the proposed extension to the Chaos Engineering approach will enable a deeper insight into the resilience of software systems? Do you consider such an extension meaningful?
4. How do you practice Chaos Engineering and how could the proposed approach be integrated into your workflow?
5. In what kind of situations would you use the proposed approach?
6. How often would you use the proposed approach?
7. Would you use parts of the approach to verify transient behavior requirements outside the scope of Chaos Engineering, i.e., as a stand-alone analysis or coupled with other types of analysis?
8. Do you think that the concept regarding the transient behavior specification part is sufficient and complete? If not, what ideas would you include?
9. Do you think that the concept regarding the transient behavior verification part is sufficient and complete? If not, what ideas would you include?
10. Currently the approach considers only event traces with discrete timestamps, i.e. at certain time points, e.g. every second/minute/hour. Does this affect you and do you consider it a weakness of the approach?

Technical questions:

The purpose of this category is to find out what the interviewees think about some of the technical aspects and properties of the prototype. The category contains the questions below.

11. Do you think that the outputs of the approach are informative enough? Would you like additional outputs to be included? If yes, what would they be?

12. Do you think that a figure, such as a line plot including the measurement data and the evaluation results, is a meaningful addition to the outputs of the proposed approach? How important would that be for you and would you use it frequently?

13. Which one of the following options would you prefer for the input of the formal behavior specification:

    13.1. Use an MTL formula as the input. This would first require you to create the formula on your own or with the support of the PSPs and tools like PSP Wizard and then to transform it into the textual MTL syntax as shown in the presentation. This option would allow the coverage of a wider range of behaviors but it would also require you to get familiar with MTL.

    13.2. Use a PSP defined as a simple text sentence as the input. This would only require you to select a pattern from a predefined pattern catalog and to provide information for the events in the pattern. This option does not require you to get familiar with MTL but the behavior coverage would be limited to the selection of supported patterns.

14. Do you think that the JSON inputs are expressive enough and contain all the necessary information? If not, what would you like to be added?

15. What do you think about the predicate inputs and functions? What predicate functions would you like to be included?

16. From what sources should the approach be able to retrieve monitoring data?

    16.1. What monitoring techniques and tools do you use and how do they preserve the monitored data?

    16.2. How would you like these to be integrated into the approach?

    16.3. Do you have suggestions for additional monitoring techniques and tools that should be integrated?

17. Should the proposed approach and its proposed tooling be integrated with other existing approaches and tools?

Non-functional requirements:

The purpose of this last category of questions is to find out whether the interviewees have some non-functional requirements regarding the prototype presented to them. The category consisted of the following questions:

18. Do you think that Usability is an important non-functional requirement for the proposed approach and prototype? How could the Usability of the approach and the prototype be improved?
19. Do you think that Extensibility is an important non-functional requirement for the proposed approach and prototype? Would you like to be able to easily extend parts of the prototype, e.g. the predicate functions or the integration with additional monitoring approaches?
20. Do you think that Performance is an important non-functional requirement for the proposed approach and prototype? What would be an acceptable time until you get a result? What size of systems should the approach be able to handle? What monitoring duration, i.e. trace lengths should be supported?

3.3 Results

The results of the three interviews for each category of questions are presented below in the respective section.

3.3.1 Introductory questions

All three of the interviewees stated that they have experience with Chaos Engineering which they rated on average as 7.2 out of 10, however, two of the interviewees said that they are lacking a bit of practical experience. Two out of the three interviewees have experienced situations in which they were not able to define the behaviors that they wanted to verify during Chaos Experiments. All of the interviewees have experienced situations in which they wanted to verify behaviors that were not concerning the steady state. Two of the interviewees were able to verify these behaviors, however, one of them had to do a lot of manual work to achieve this, and the third interviewee was able to achieve this only partially.

3.3.2 Conceptual questions

All of the participants think that the proposed extension to the Chaos Engineering approach will provide a deeper insight into the resilience of software systems. In response to the question asking about how the extended approach could be adopted by the interviewees, one of them said that he was not actively practicing Chaos Engineering at the moment so he could not answer properly. Another interviewee answered that some interfaces or adapters would be required to integrate the extended Chaos Engineering approach into his workflow. The last interviewee answered that integration into his workflow would not be difficult since he uses similar technologies as the initial prototype. Two out of the three interviewees answered that they would regularly use the proposed approach if their

Chaos Experiments have some defined criteria regarding the transient behaviors of the system. The third interviewee answered that he would use the proposed approach in situations where he would like to know more about the system than just if a certain service is still running. Additionally, two of the interviewees said that they could imagine using the approach and the prototype for scenarios and use cases outside the scope of Chaos Engineering and Chaos Experiments. When asked whether they think the concept regarding the specification of the transient behaviors is complete and sufficient, all of them answered positively, however, two of them expressed the concern that extracting meaningful values for the specifications might be difficult. Additionally, two of the participants said that some form of assistance for the creation of the transient behavior specifications would be quite helpful. When the interviewees were asked whether they think that the concept regarding the verification of the specifications is complete and sufficient, they again answered mostly positively, only making remarks that including more information into the output of the verification such as which parts of the behavior specification failed, time intervals, and differences would be useful. Regarding the question enquiring whether the interviewees view the discrete nature of the event traces as a limitation of the approach, two of the participants said that this could be a limitation of the approach whereas the third interviewee said that in practice this is a strength of the approach since he is not aware of monitoring tools that capture measurement data in non-discrete ways.

3.3.3 Technical questions

When asked if they thought the results of the approach were informative enough, all interviewees added that additional information such as information about the time intervals and frames as well as the value deviations would be useful. Two of the interviewees said that a plot visualizing the monitoring data and the verification results would be a useful and important addition to the prototype, while the third participant said that this feature would be nice to have but not very important. When presented with the different input options for specifying transient behavior, namely an MTL formula or a PSP, all interviewees said they would like to have both for different reasons. Some of the main arguments in favor of PSPs are that it is generally quicker to create a specification with them and that they are more practical and readable, especially when a person is working with specifications that they did not create themselves. An argument in favor of MTL is that it could enable more precise and detailed specifications by using the full extent of the logical formalism. Regarding the JSON fields concerning the behavior specification, the interviewees only suggested additional logical predicates, such as increasing and decreasing trends and predicate functions supporting Boolean values occurring in the event traces. Regarding any monitoring tools that can be integrated into the approach, one of the interviewees could not think of any tools to be integrated, another interviewee said that he is currently using Comma-Separated Values (CSV) files to capture the events in his system, and the last one suggested supporting time series databases in general, and Prometheus in particular.

3.3.4 Non-functional requirements

One of the interviewees expressed the opinion that usability is the most important non-functional requirement for the prototype and that other features could even be neglected in favor of usability. Two of the interviewees also said that a user interface would increase the usability of the prototype. Additionally, two of the interviewees also said that having a validation of the JSON inputs against a

JSON schema would also increase the usability. Furthermore, all interviewees said that extensibility is also an important feature of the prototype and would like to be able to extend parts of it if necessary. Regarding performance, two interviewees said that the response time of the verification is not of utmost importance and that the event trace lengths should not be very long as the transient behaviors normally occur for a shorter amount of time, e.g. couple of minutes.

3.3.5 Interpretation

In this section, the results of the interviews are interpreted and analyzed with regard to the hypotheses defined in Section 3.2.2 which are subsequently confirmed or rejected. The results regarding the hypotheses will also decide what will be included in the final concept of the extended Chaos Engineering approach.

The answers to the introductory questions confirm hypotheses H_1 and H_2, as the majority of participants had already experienced difficulties while defining the behaviors they wanted to verify by using Chaos Engineering and also while analyzing transient behaviors with it. These results further solidify the motivation of the thesis and the need for an extension of the Chaos Engineering approach that would allow an analysis of the transient behaviors. Furthermore, all of the interviewees consider such an extension meaningful which as a result confirms hypothesis H_3.

A majority of the participants indicated that they would use the approach and prototype for other scenarios outside of Chaos Engineering. This confirms hypothesis H_4 and as a result, introduces the requirement that the approach and its implementation should be able to provide verification of transient behavior requirements to other tools and approaches outside of Chaos Engineering. The approach for specifying transient behavior requirements using MTL and PSPs was considered sufficient and complete by all interviewees. As a result, this confirms hypothesis H_5 and leads to a requirement for the final concept of the extended Chaos Engineering approach, which is to support both MTL and PSPs for the specification of the behaviors. Furthermore, the interviewees also considered the concepts regarding the verification of the transient behavior requirements and in particular, the usage of runtime verification as sufficient and complete, which as a result confirms hypothesis H_6 and results in the requirement to integrate runtime verification monitors into the final concept for the approach. Additionally, since a majority of the interviewees also stated that a visualization of the verification results and the monitoring data in the form of a plot is an important addition to the approach and the prototype, hypothesis H_7 is also confirmed and the inclusion of a visualization is also considered as a requirement for the final concept. The interviewees also stated that the non-functional qualities of usability, extensibility, and performance are important for the final concept and the resulting prototype, and therefore also hypothesis H_8 is confirmed. Since a majority of the interviewed people said that the introduction of a graphical user interface would significantly increase the usability of the prototype, the introduction of such is included as a requirement for the final concept and the implementation of the prototype.

4 Extended Chaos Engineering Approach

After identifying initial concepts that could be used to extend the Chaos Engineering approach to support the verification of transient behavioral specifications from literature, and after conducting the interviews with the experts, enough requirements have been collected so that they can now be combined into a final concept for the extended Chaos Engineering approach.

As defined by Rosenthal and Jones [RJ20], the steps in the Chaos Experimentation process are as follows: (1) build a steady-state hypothesis, (2) vary real-world events, (3) run experiments in production, (4) automate experiments, (5) minimize blast radius. Since the Chaos Engineering approach is extended, also the above-mentioned steps have to be extended accordingly. A newly introduced step is the creation of the transient behavior hypothesis and its place in the extended Chaos Experimentation process is right after the creation of the steady-state hypothesis. As a result, the extended Chaos Experimentation process is defined as follows: (1) build a steady-state hypothesis, (2) build a transient behavior hypothesis, (3) vary real-world events, (4) run experiments in production, (5) automate experiments, (6) minimize blast radius.

Furthermore, the characteristics of the transient behavior hypothesis and its differences from the steady-state hypothesis have to be defined. Traditionally, the steady-state hypothesis of an experiment is a simple statement regarding a single measurement of the system at a certain time point, which is verified against a certain tolerance [Mil19]. Even if the steady-state hypothesis is carried out continuously during an experiment, it is still a statement regarding only a single measurement at a single point in time and its acceptance criteria can be inherently different from those of the transient states. In contrast, the transient behavior hypothesis does not refer to a single point in time, but to a series of points in time or even to the entire experiment in which the transient states and behaviors of the system occur. Therefore, the transient behavior hypothesis must be able to define statements over longer time periods and not only a single point in time like the steady-state hypothesis. Furthermore, the transient behavior hypothesis should also be able to define statements regarding multiple measurements and metrics of the system since the transient behaviors and states of a system are rarely affecting a single component or service. These metrics must be able to cover properties that can be observed either from the outside of the system, e.g. response time, or the inside, e.g. number of active instances, utilization of the instances, etc. As the transient behavior hypothesis can concern multiple measurements there must also be a way to connect and compare them in a formal way. Similarly to the steady-state hypothesis verification, also the verification of the transient behavior hypothesis should produce a single Boolean result.

In addition to extending the Chaos Experimentation process, also the elements and concepts of the Chaos Engineering approach have to be extended. Traditionally these include a Chaos Experiment consisting of a steady-state hypothesis, an anomaly that is injected, and a test system on which the experiment is conducted. Even though the Chaos Experimentation steps that are listed above can be executed manually, the usage of tools that can support software engineers in executing experiments is particularly helpful also for the step which regards the automation of the Chaos Experiments [RJ20]. Furthermore, various research has also successfully applied the principles of Chaos Engineering by using a Chaos Engineering tool to automate the execution of experiments [FHW+21; KHFH20]. Therefore the usage of a Chaos Engineering tool is considered a part of the traditional Chaos Engineering approach. Next, the traditional Chaos Engineering approach is extended similarly to how the steps of the Chaos Experimentation process are extended. The extended approach introduces several new elements to the already existing ones. These new elements are a transient behavior hypothesis, a transient behavior verification, a transient behavior monitoring of the system's events, a visualization of the results, and a Graphical User Interface (GUI). The purpose of the transient behavior hypothesis is to make testable statements regarding the system under test as described above. The purpose of the transient behavior validation is to provide a concrete result regarding whether the behavior hypothesis holds or is rejected. The goal of the visualization of the result is to present the results in an intuitive and readable manner to the users of the approach as requested by the interviewees. As elicited in the expert interviews, the extended approach should also be usable as a stand-alone analysis without the need of a Chaos Engineering tool or Chaos Engineering itself, therefore the extended approach also includes a GUI providing this stand-alone analysis. Figure 4.1 depicts an overview of the extended Chaos Engineering approach, including the already existing and the newly introduced elements and the interactions between them.

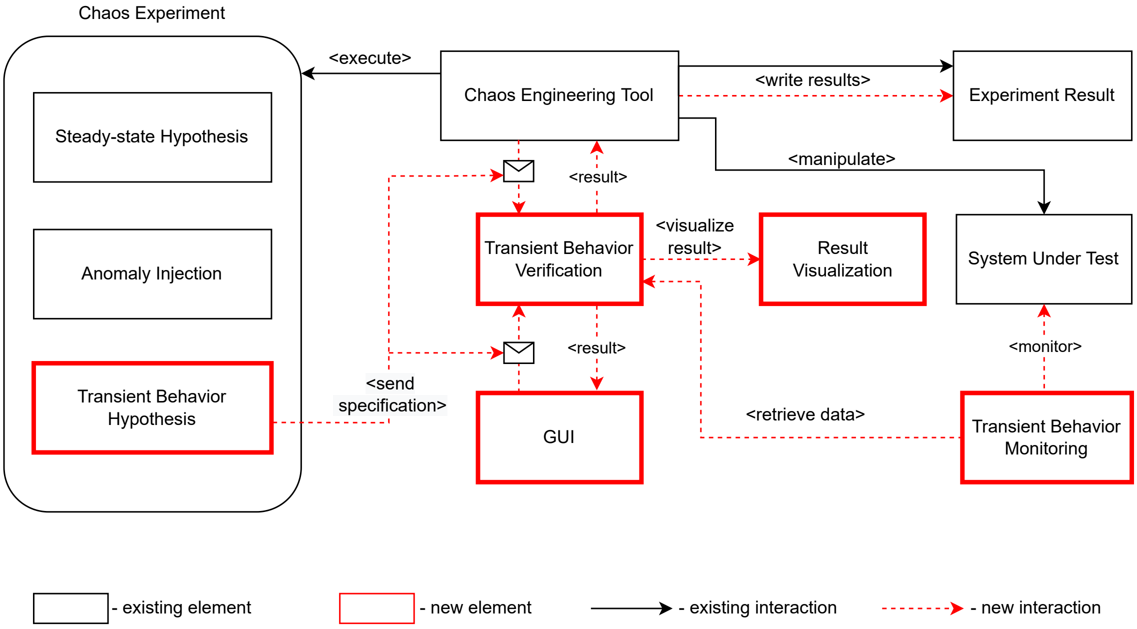


Figure 4.1: Extended Chaos Engineering Approach - Concept Overview.

4.1 Transient Behavior Hypothesis Specification

The first element introduced by the concept is the transient behavior specification section. The specification section is introduced as an additional step into the Chaos Experiment since it is conceptually different from the steady-state hypothesis and a separate step is where it inherently

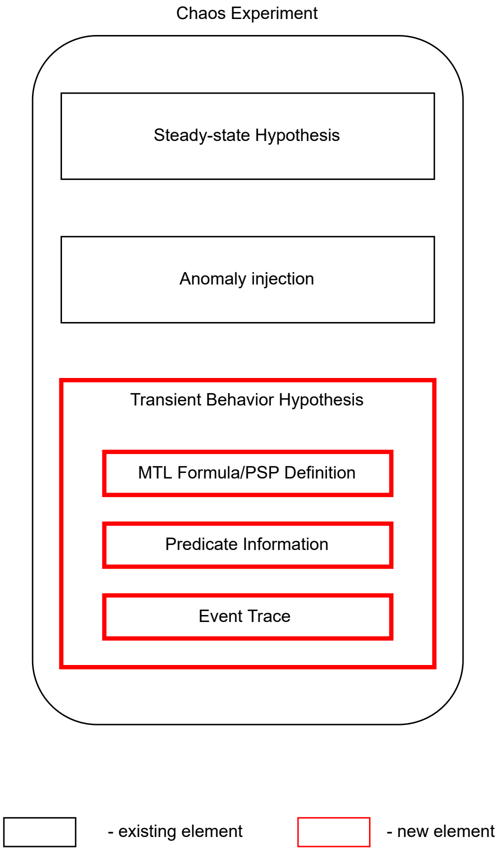


Figure 4.2: Transient Behavior Hypothesis Specification Concept.

belongs. The specification must contain a formal description of the transient behavior in order to give it a specific meaning and interpretation, and to make it compatible with approaches like runtime verification. As a result, the specification must use a formalism capable of describing the temporal properties of the transient behaviors, for which also runtime verification monitors have already been constructed. Such formalisms were already introduced in Chapter 2, namely MTL and the PSPs which can be seen as an abstraction layer, abstracting the complexity of the logical syntax of the different temporal logics. Additionally, including PSPs allows the users of the approach to also use tools like the PSP Wizard by Grunske et al. [AGL+15] in order to easily create PSP definitions, which as a result would also increase the usability of the proposed approach. Furthermore, the approach and the framework proposed by Grunske et al. [AGL+15] have already been successfully applied to formalize the requirements of an industrial case study as PSP definitions. As a result, the PSPs are also considered viable for the definition of the transient behavior hypotheses. Therefore, the transient behavior specification must contain a formal behavior description specified as either an MTL formula or a PSP definition. In addition, since the logical variables contained in an MTL formula or a PSP definition do not include information regarding how they should be interpreted, the behavior specification must also contain information concerning the logical variables, events, and predicates featured in the behavior description. To illustrate this, consider the following MTL formula: $\blacksquare(A \rightarrow \blacksquare(B))$. In order to enable an evaluation of the formula to either $true$ or $false$, the logical variables A and B and their assignments have to be defined. As already mentioned, these assignments have to be comparisons concerning the event traces and the measurement data, and they should be easily mappable to Boolean values, i.e. $true$ or $false$. Some example assignments are comparisons of the response time or the instance count against a certain threshold. Furthermore, the behavior specification requires the specification of an event trace of the behaviors occurring in the system, in order to enable the runtime verification of the specified behavior. Figure 4.2 depicts a Chaos Experiment containing a transient behavior specification including all newly introduced elements.

4.2 Transient Behavior Verification

The second element required by the concept for the extended Chaos Engineering approach is the verification of the transient behavior hypothesis. Firstly, the verification must be able to be triggered and to receive transient behavior requirements from the Chaos Engineering tool and also from the GUI. Furthermore, the verification must be able to return its results to both the Chaos Engineering tool and the GUI. As elicited in the expert interviews, the verification must be able to return a Boolean result whether the transient behavior hypothesis holds. The purpose of such a concrete Boolean value regarding the result of the verification is the fact that it provides a simple means to understand whether the hypothesis holds or not. A further argument for the inclusion of such a result is the fact that also the result of the steady-state hypothesis is a concrete Boolean value in the traditional Chaos Engineering approach. Nevertheless, as requested by the interviewees, additional outputs regarding the result of the verification should also be included with the Boolean value. As elicited, this additional information includes more detailed information regarding the different time intervals in the verification, such as the time points in which the result changes from one value to another, e.g. from $true$ to $false$, and the values that caused this change. The inclusion of these additional outputs provides the reasoning behind the final verification result and makes the outputs of the extended approach more informative altogether. In order to be able to create all of the above-discussed results and to correctly verify the transient behavior specifications, the verification must include the usage of an approach such as runtime verification [LS09]. Since this requires also the usage of monitoring data regarding the system's states, the transient behavior verification must also be able to retrieve such from a transient behavior monitoring approach.

4.3 Verification Result Visualization

As requested by the interviewees, the above-discussed results of the verification are also presented visually in the form of a plot which also includes the event traces used in the verification in order to make the interpretation of the results easier. The inclusion of a graphical presentation of the measurement data provides further reasoning behind the results of the verification without the need to manually retrieve and examine the measurement data which is often contained in unintuitive ways in the event traces. Additionally, the visualization of the measurement data and the verification results may enable further analyses or comparisons of the data as it contains more information than the Boolean results returned by the verification. The proposed plot for the visualization is a line chart that includes the event traces used during the verification presented as lines and the results of the verification are indicated by the background colors of the respective section in the plot.

4.4 Graphical User Interface

A majority of the interviewees said that an inclusion of a GUI that supports the user in creating transient behavior hypothesis specifications and the insertion of these specifications into the Chaos Experiments would significantly increase the usability of the proposed approach. Furthermore, a majority of the interviewees also requested that the approach should also be usable outside the scope of Chaos Engineering and Chaos Experiments. Therefore, in order to provide these, the concept for the transient behavior verification also includes a GUI that enables and supports the users while specifying transient behavior hypothesis specifications, their insertion into Chaos Experiments, and the stand-alone execution of transient behavior verifications.

4.5 Transient Behavior Monitoring

Even though transient behavior monitoring is considered in this concept, this thesis is focusing only on the specification and verification of transient behaviors. As a result, the concept simply introduces the usage of such an approach and assumes that it is capable of monitoring the events occurring in the system under test. Additionally, the concept assumes that the monitoring approach is monitoring and measuring the occurring events in a discrete-time manner and the monitored event traces can be easily retrieved by the transient behavior verification. Particularly suitable for this purpose are time series databases because of the way they store data. Furthermore, since there may be various monitoring approaches that provide the monitored data in different formats and ways, there should be a common format that is supported by the concept, in order to abstract the differences that the various monitoring approaches may have. Such a common format are CSV files.

4.6 Interaction of the Proposed Concepts

The purpose of this section is to describe how all elements interact with each other when a Chaos Experiment containing a transient behavior hypothesis specification is executed using the extended Chaos Engineering approach proposed in this chapter. First, the initial steady-state check and the anomaly injection get executed as in the traditional Chaos Engineering approach. After the anomaly has been injected, enough time has passed to let the transient behaviors occur, and the second steady-state check has been performed, the transient behavior check will be initiated. For this, the Chaos Engineering tool will trigger the verification and will also send the transient behavior specification to it. After the transient behavior specification has been received, the monitoring data required for the verification of the specification is retrieved, and afterward, the specification is prepared for verification. After the verification is complete, the results and the monitoring data are plotted and the results are returned from the verification to the Chaos Engineering tool. This whole process should take at most a couple of seconds and once it is completed, the Chaos Engineering tool should have a clear Boolean result regarding the verification of the transient behaviors, i.e. it should have received a clear $true/false$ result. Figure 4.3 depicts the above described interaction as a UML sequence diagram.

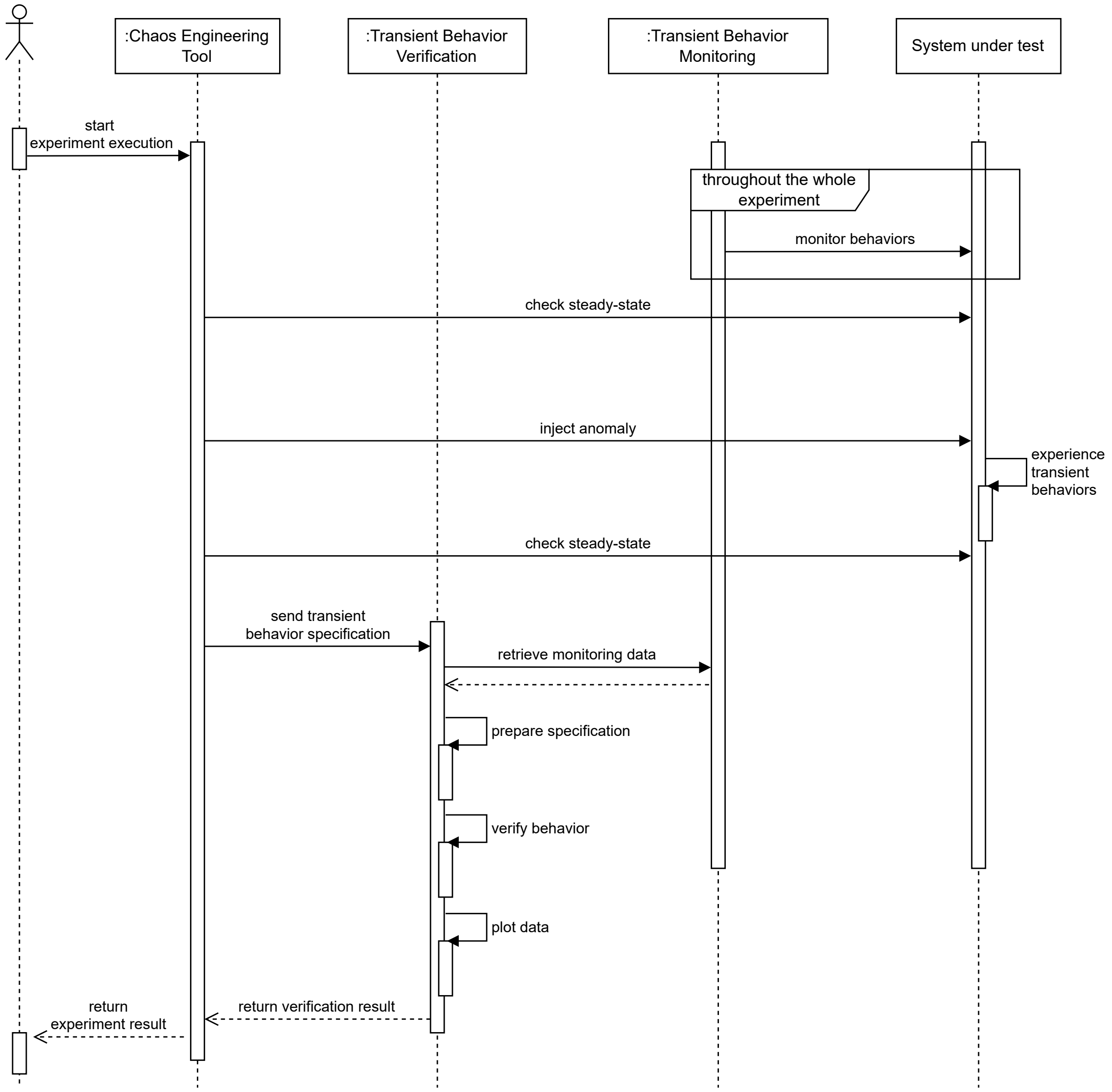

:Chaos Engineering Tool
:Transient Behavior Verification
:Transient Behavior Monitoring
System under test
start experiment execution
throughout the whole experiment
monitor behaviors
check steady-state
inject anomaly
experience transient behaviors
check steady-state
send transient behavior specification
retrieve monitoring data
prepare specification
verify behavior
plot data
return verification result
return experiment result

Figure 4.3: Transient Behavior Verification - Sequence Diagram.

5 Prototype Implementation

In this chapter, an overview of how the concept presented in Chapter 4 is implemented into a final functioning prototype is presented. The development of the prototype includes the selection of a Chaos Engineering tool which is responsible for executing the Chaos Experiments, the definition of a format for the transient behavior specification, the development of the transient behavior verification tool, and the integration of several time series databases used as monitoring tools. In the final implementation of the prototype, Chaos Toolkit is used as the Chaos Engineering tool of choice. The behavior verification, result visualization, and GUI are implemented as the components of a Python [Pyt] application. Because of the discrete-time nature of the transient behavior verification, the prototype requires the integration of time series databases as monitoring tools. The monitoring tools that the final prototype supports are Prometheus [Pro] and InfluxDB [Inf]. Additionally, the prototype also supports the retrieval of monitoring data from CSV files. An overview of the prototype and the items it interacts with are presented in Figure 5.1.

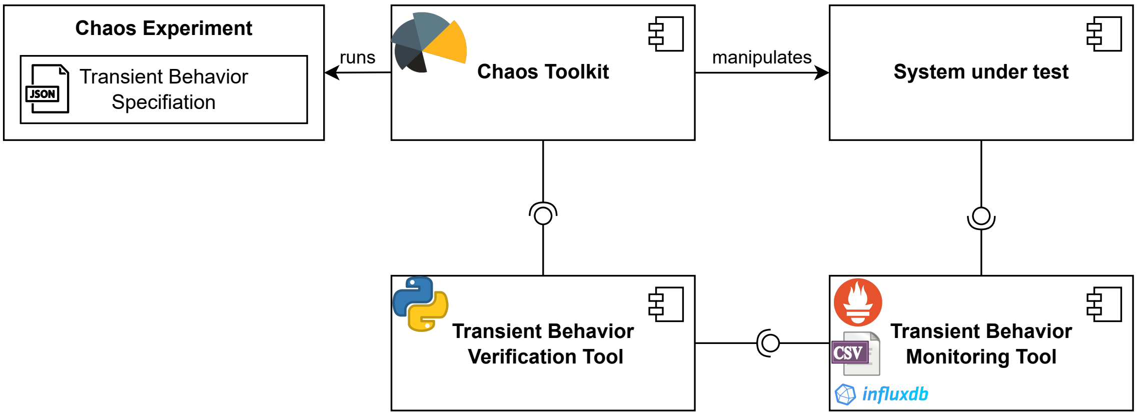

Chaos Experiment
JSON
Transient Behavior Specifiation
runs
Chaos Toolkit
manipulates
System under test
Transient Behavior Verification Tool
Transient Behavior Monitoring Tool
CSV
influxdb

Figure 5.1: Prototype Components - Overview.

5.1 Chaos Engineering Tool

The first thing required for the implementation of the concept is the selection of the Chaos Engineering tool that the prototype uses to run Chaos Experiments. For this purpose, the following are identified as possible candidates: Chaos Mesh [Chaa], Gremlin [Gre], ChaosBlade [Chad], Chaos Toolkit [Chac], Litmus [Lit], and Chaos Monkey [Chab]. Table 5.1 presents a comparison of the above-mentioned tools with regard to the different layers on which they support failure injections and whether they are open-source since those are important criteria in the selection of the tool in the final prototype implementation.

	Chaos Mesh	Chaos Monkey	Gramlin	Chaos Toolkit	Litmus	ChaosBlade
Platform layer injections	x	x	x	x	x	x
Network layer injections			x	x		
Application layer injections				x		x
Open-source	x	x		x	x	x

Table 5.1: Chaos Engineering Tools - Comparison

The first tool that is removed from the selection pool is ChaosBlade as it is lacking documentation written in English. The second tool that is removed from the selection pool is Gremlin, as it is not free-to-use. From the remaining tools, Chaos Toolkit supports the injection of failures and anomalies on more layers than the other tools. Another important factor also in favor of Chaos Toolkit is that it is the only tool that specifies Chaos Experiments in the format defined by Rosenthal and Jones [RJ20], namely containing a steady-state hypothesis and an anomaly injection. The rest of the Chaos Engineering tools either use proprietary Chaos Experiment formats that are not compatible with the one defined by Rosenthal and Jones [RJ20] or do not have any formal definition of what a Chaos Experiment contains. Furthermore, research has already shown that Chaos Toolkit is a tool capable of correctly executing Chaos Experiments [FHW+21; KHFH20]. In addition, Chaos Toolkit has already been successfully extended with an additional hypothesis validator [FHW+21], which is similar to the transient behavior verification proposed by the concept in Chapter 4. As a result, considering the above-discussed facts, Chaos Toolkit is chosen as the Chaos Engineering tool, which is used as part of the prototype.

Additionally, a communication protocol has to be selected, with which the Chaos Engineering tool has to exchange data with the verification tool. A common protocol supported by at least two of the Chaos Engineering tools, namely Chaos Toolkit and Gremlin, is the HTTP protocol. As a result, the HTTP protocol is selected for the communication between the Chaos Engineering tool and the verification tool.

5.2 Transient Behavior Specification

As already mentioned in the previous chapter, the specification should contain a transient behavior specified as an MTL formula or a PSP definition together with information regarding the behavior's logical predicates and the measurement data. In addition, the format that is to be selected should be compatible with the HTTP protocol as this is how the specification will be sent to the verification tool. As a result, the technology and the format selected to contain the above-mentioned information is a JSON object. In the following, each field of the JSON object is listed and explained.

behavior_description: An optional text field that allows the specification of a text description of the transient behavior.

specification: A required text field containing the formal description of the behavior, which is either an MTL formula, where the symbols for the temporal operators are replaced with their names or a textual PSP definition created using the PSP Wizard tool [AGL+15].

specification_type: A required text field specifying the type of the formal behavior specification, can be either “mtl“ or “psp“.

future-mtl: An optional field allowing to specify that a future-MTL formula has been defined in the “specification“ field. By default, the prototype assumes that the specified behaviors are in past-MTL. Additionally, the field should only be used when “specification_type“ is set to “mtl“.

predicates_info: A required array containing the information regarding the logical predicates in the formal behavior specification. The array contains objects of the following format:

- **predicate_description:** An optional text field that allows the specification of a text description of the logical predicate.
- **predicate_name:** A required text field containing the predicate name as it occurs in the formal specification above.
- **predicate_logic:** A required text field containing the name of a logical function from a predefined selection of Boolean operations provided by the prototype.
- **predicate_comparison_value:** A text field containing a value used as the comparison value for the respective logical operation. Not all functions require the inclusion of this field, e.g., the “boolean“ function and the trend functions.

measurement_source: A required text field defining the source of the measurement data, currently supporting “influx“ for InfluxDB, “prometheus“ for Prometheus,“csv“ for local CSV files, and “remote-csv“ for remote CSV files, for example hosted on a web server.

remote-csv-address: Required when “measurement_source“ is set to “remote-csv“. This field contains the URL to the CSV table.

measurement_points: A required array containing information regarding the measurement data.

- **measurement_name:** A required text field containing the name of the measurement as it occurs in the behavior specification.
- **measurement_query:** A text field specifying the query that will be used to retrieve the data. Only required when “measurement_source“ is specified as “influx“ or “prometheus“.
- **measurement_column:** A text field specifying the column from which the measurement data will be retrieved. Only required when “measurement_source“ is set to “csv“ or “remote-csv“.
- **start_time:** A text field for specifying the start time of the query interval. Only required when “measurement_source“ is set to “prometheus“.
- **end_time:** A text field for specifying the end time of the query interval. Only required when “measurement_source“ is set to “prometheus“.
- **steps:** A text field for specifying the steps of the query interval. Only required when “measurement_source“ is set to “prometheus“.

Listing 5.1 Transient Behavior Specification - MTL & CSV Example.

```
{
    "behavior_description": "When the response time of a service changes from normal to high, it can remain high for at most 5 time units.",
    "specification": "( resp_time_high(resp_time) since[0,5] resp_time_stable(resp_time) )",
    "specification_type": "mtl",
    "predicates_info": [
        {
            "predicate_name": "resp_time_stable",
            "predicate_description": "Response time is stable below 0.5",
            "predicate_comparison_value": "0.5",
            "predicate_logic": "smallerEqual"
        },
        {
            "predicate_name": "resp_time_high",
            "predicate_description": "Response time is unstable above 0.5",
            "predicate_comparison_value": "0.5",
            "predicate_logic": "bigger"
        }
    ],
    "measurement_source": "csv",
    "measurement_points": [
        {
            "measurement_name": "resp_time",
            "measurement_column": "resp_time"
        }
    ]
}
```

Listing 5.1 presents an example transient behavior specification using an MTL formula as the formal specification and a CSV file as the source of the event traces. The formula contains two logical predicates, namely `resp_time_high` and `resp_time_stable`. The `resp_time_high` is *true* when the `resp_time` measurement is bigger than 0.5 and the `resp_time_stable` is *true* when the `resp_time` measurement is equal to or smaller than 0.5. The specification also defines that the `resp_time` measurement is retrieved from the table column with the same name.

Listing 5.2 presents a transient behavior specification using a PSP definition. The pattern used in the specification contains the `resp_time_high` and `instance_increase` predicates. The `resp_time_high` predicate evaluates to *true* when the `resp_time` measurements are greater than or equal to 100 and the `instance_increase` predicate evaluates to *true* when the `instance-count` measurements are strictly increasing. The transient behavior specification also defines the source of the `resp_time` and `instance_count` measurements, which in this case is an InfluxDB instance, and the queries used to retrieve the individual measurements.

Listing 5.2 Transient Behavior Specification - PSP & InfluxDB Example.

```
{
    "behavior_description": "When the response time exceeds 100 ms, a new instance of the
↪ service should be spawned.",
    "specification": "Globally, if {resp_time_high(resp_time)} [has occurred] then in
↪ response {instance_increase(instance_count)} [eventually holds] between 30 and 60 time
↪ units.",
    "specification_type": "psp",
    "predicates_info": [
        {
            "predicate_name": "resp_time_high",
            "predicate_description": "Response time is higher than 100ms",
            "predicate_comparison_value": "100",
            "predicate_logic": "biggerEqual"
        },
        {
            "predicate_name": "instance_increase",
            "predicate_description": "Instance counts increase",
            "predicate_logic": "trendUpwardStrict"
        }
    ],
    "measurement_source": "influx",
    "measurement_points": [
        {
    "measurement_name":"resp_time",
    "measurement_query":"SELECT \"AvgResponseTime\" FROM
↪ \"TimeBatchRuns\".\"autogen\".\"Batch_Time\""
    },
    {
    "measurement_name":"instance_count",
    "measurement_query":"SELECT \"counts\" FROM
↪ \"AliveInstances\".\"autogen\".\"InstanceCounts\""
    }
    ]
}
```

As already mentioned and shown in Listings 5.1 and 5.2, the prototype supports behavior definitions using MTL formulas and PSPs. When a PSP behavior specification is received by the verification tool, the PSP behavior definitions are translated into past-MTL formulas using already existing mappings for each pattern. At the time of writing, the supported patterns for which there are mappings into past-MTL are the following: Absence, Universality, Recurrence, and Response. For these patterns, a simple natural language definition as shown in Listing 5.2 can be used. For any other PSP that is not supported or that should be mapped into future-MTL, the MTL formula resulting from the mapping of the PSP to MTL should be used instead. One approach allowing an easy mapping of PSPs into future-MTL formulas is the PSP Wizard tool by Grunske et al. [AGL+15]. Additionally, the Boolean functions that can be assigned to the logical predicates in the behavior specifications are also limited to a certain selection. At the time of writing, the prototype supports simple numerical comparisons such as “`bigger`“, “`biggerEqual`‘, “`smaller`“, “`smallerEqual`“,“`equal`“, and trend functions such as “`trendUpward`“, “`trendUpwardStrict`“, “`trendDownward`“, “`trendDownwardStrict`“. Even though the selection regarding the PSPs and the logical functions is limited, these are implemented in a way that allows easy extensions when required. The extension of the supported PSPs requires only the addition of a regular expression for the textual representation of the pattern in the file containing the remaining such and the MTL formula to which the pattern is mapped in another class containing the mappings. The extension of the Boolean functions can be done by simply adding further Python functions that return a Boolean value in the respective class containing these.

5.2.1 Transient Behavior Specifications in Chaos Experiments

The transient behavior specifications shown above now have to be incorporated into Chaos Experiments. As Chaos Toolkit is the Chaos Engineering tool of choice for the prototype, the Chaos Experiments are defined using Chaos Toolkit's format and the transient behavior specification is adapted to be compatible with Chaos Toolkit. Furthermore, in order to send the behavior specification to the verification component using the HTTP protocol, the transient behavior specification in the Chaos Experiment is defined inside of a probe that uses the HTTP provider. This probe is placed in the steady-state hypothesis section of the experiment even though the transient behavior specification is conceptually different, because any probes outside of the steady-state hypothesis, for example in the method, do not check the predefined tolerances. Although this is a conceptual limitation of the Chaos Experiment and Chaos Toolkit, this does not affect the execution of the experiment and the verification of the transient behavior specification. As already mentioned, the probe uses the HTTP provider of Chaos Toolkit, which requires the specification of some additional fields, for example, the URL of the endpoint, the request method, and information regarding headers among others. The tolerance of the probe is set to the expected verification result. An example of a Chaos Experiment containing a transient behavior specification using Chaos Toolkit's format can be seen in Listings 5.3 and 5.4. Listing 5.3 presents the steady-state hypothesis and the method of the experiment, however, without the probes of the steady-state hypothesis because of size restrictions. Thus, the probes are displayed in Listing 5.4. In the presented experiment, the steady-state hypothesis of the experiment is that the payment deployment is fully available. The failure that is injected is a termination of one of the payment pods. The failure is introduced at the 30th second of the experiment and the verification of the steady-state hypothesis and the transient behavior hypothesis are carried out 90 seconds after that, i.e. at the 120th second. The transient behavior hypothesis is specified using a PSP and its meaning is that after the response time has been stable, it is never the case that it becomes unstable.

Listing 5.3 Chaos Experiment Containing a Transient Behavior Specification - Steady-State Hypothesis (Without Probes) and Method. (Continued in Listing 5.4)

```
{
   "title":"Terminate one payment pod",
   "description":"Terminating one payment pod and observing steady-state hypothesis and transient behavior",
   "tags":["kubernetes"],
   "steady-state-hypothesis":{
      "title":"Verifying that deployment remains healthy",
      "probes":[...]
   },
   "method":[
      {
         "type":"action",
         "name":"terminate-payment-service-pod",
         "provider":{
            "type":"python",
            "module":"chaosk8s.pod.actions",
            "func":"terminate_pods",
            "arguments":{
               "name_pattern":"payment",
               "rand":true
            }
         },
         "pauses":{
            "before":30,
            "after":90
         }
      }
   ]
}
```

Listing 5.4 Chaos Experiment Containing a Transient Behavior Specification - Probes in the Steady-State Hypothesis (Continuation of Listing 5.3).

```
{
   ...
   "probes":[
      {
         "name":"deployment-is-fully-available",
         "type":"probe",
         "tolerance":true,
         "provider":{
            "type":"python",
            "module":"chaosk8s.probes",
            "func":"deployment_fully_available",
            "arguments":{
               "name":"payment"
            }
         }
      },
      {
         "name":"check-transient-behaviors",
         "type":"probe",
         "tolerance":{
            "type":"jsonpath",
            "path":"$.result",
            "target":"body",
            "expect":[
               "True"
            ]
         },
         "provider":{
            "type":"http",
            "url":"http://localhost:5000/monitor",
            "method":"POST",
            "headers":{
               "Content-Type":"application/json"
            },
            "arguments":{
               "behavior_description":"description",
               "specification":"After {resp_time_stable(resp_time)}, it is never the case that {resp_time_unstable(resp_time)} [holds].",
               "specification_type":"psp",
               "predicates_info":["..."],
               "measurement_source":"influx",
               "measurement_points":["..."]
            }
         }
      }
   ]
   ...
}
```

5.3 Transient Behavior Verification Tool

The verification tool was implemented in Python since the approach for creating runtime verification monitors for MTL formulas by Ulus [Ulu19b] that was used as part of the prototype is implemented as a Python library [Ulua] and also since Chaos Toolkit is implemented in Python. An overview of the underlying components of the verification tool is presented in Figure 5.2.

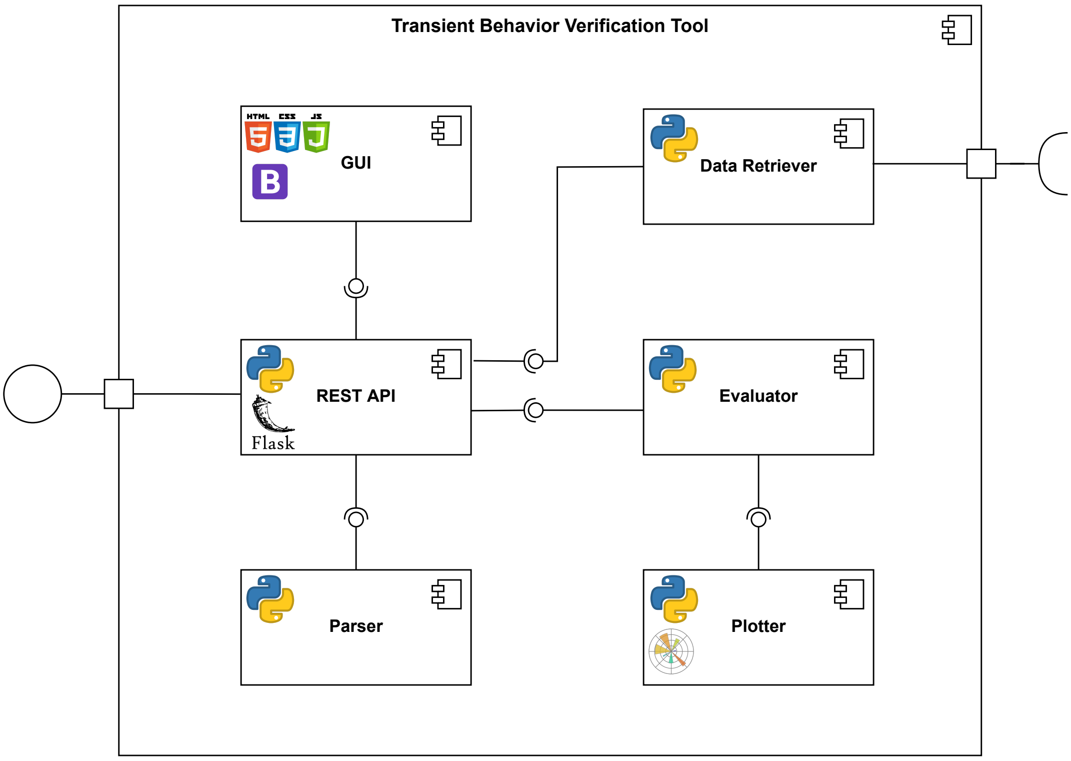


Figure 5.2: Component Diagram of the Verification Tool.

REST API

To enable communication via the HTTP protocol, a REST API is implemented with the help of the web framework Flask [Fla]. There are several endpoints provided by the API, however, the most important one is the “`/monitor`“ endpoint. This endpoint is listening for HTTP requests containing transient behavior specifications as JSON objects and when such a request is received, the verification of the received specification is initiated. The REST API is also used to serve the web pages comprising the graphical user interface.

Parsing Component

The verification tool also contains a component that is responsible for the mapping of PSP specifications into MTL formulas and the preparation steps for the verification of the formula, which also includes the processing of the information regarding the logical predicates in the specification.

The Boolean functions corresponding to the “predicate_logic“ field from the behavior specification are implemented as the functions of a Python class located namely in this component. In Listing 5.5, a selection of some of the Boolean functions and the constructor of the class are presented. The value of the class attribute called “value“ which is assigned in the constructor is retrieved from the “predicate_comparison_value“ field in the specification. As shown in the listing, the implementation of the predicate functions is kept as simple as possible in order to allow easier extension and addition of further functions when required.

Listing 5.5 Predicate Functions.

```
class Predicates:
    def __init__(self, value=None):
        self.value = value
        self.trendLast = None
    def equal(self, variable):
        return variable == self.value
    def notEqual(self, variable):
        return variable != self.value
    def bigger(self, variable):
        return variable > self.value
    def smaller(self, variable):
        return variable < self.value
    ...
```

Data Retrieval Component

Another component also a part of the verification tool is the data retriever component, which consists of one abstract class and three subclasses extending it, responsible for the retrieval of the monitoring data. At the time of writing, there are three data retriever subclasses, namely for InfluxDB, Prometheus, and CSV files. These three classes also have the same output format, namely a multidimensional array, where each column corresponds to a measurement point from the behavior specification. As a result, also the addition of further data retriever classes for different monitoring tools is kept as simple as possible.

Evaluation Component

The verification tool includes an additional evaluation component, responsible for the verification of the prepared MTL formula against the monitoring data. This is achieved by utilizing a Python library [Ulua] that creates the runtime verification monitors according to the approach of Ulus [Ulu19b]. Even though at the time of writing the used library is deprecated in favor of a newer implementation [Ulub] in C++, the old implementation is preferred due to several factors. First and most important, the input format of the MTL formulas in the Python implementation is more suitable for the needs of this thesis. This concerns in particular the definition of the predicates in the formulas. The Python library allows the binding of the predicates in the formula to Boolean functions as already described, while the newer implementation does not support this as it supports only a limited number of predefined functions. As a result, the choice of the Python implementation allows the definition of more complex logical predicates and also their extension and addition of further ones in the future. Another factor is that the newer implementation is realized in C++ and even though the monitors are accessible from Python via bindings, the original Python implementation is preferred as it is written in Python and ensures consistency with the remainder of the verification tool which is also implemented in Python. Furthermore, the claimed performance improvements in the newer implementation are not critical for the prototype since as elicited in the interviews, the response time of the verification is not of critical importance. Considering all of the above, the original Python implementation is deemed sufficient for the purpose of this thesis.

The creation of the monitors and the verification of the behaviors is initiated by simply calling the methods provided by the imported library. During the verification, the monitor returns a Boolean value for each time point from the event trace. These results are collected in order to calculate the intervals in which the behavior specification was either satisfied or violated. The result regarding the last time point is accepted as the final result of the verification and is later returned as the final Boolean value. Additionally, the information regarding the intervals is passed to the plotting component in order to create the visualization of the verification results.

Furthermore, the approach of Ulus [Ulu19b] supports only past-MTL formulas. In order to also allow support for future-MTL formulas, some adaptations have to be introduced. To make a future-MTL formula verifiable by the past-MTL monitors, the future temporal operators in the formula have to be replaced with their past variants and the event trace has to be processed in reverse, i.e. starting from its end. To illustrate this, consider the following two PSPs and their mappings in respectively past and future-MTL.

PSP	After $\{A\}$, it is always the case that $\{B\}$ [holds] for 5 time units.
Past MTL Formula	$\blacksquare(\blacklozenge_{[0,5]} A \rightarrow (B \mathcal{S} A))$
Future MTL Formula	$\square(A \rightarrow \square_{[0,5]} B)$
PSP	Before $\{A\}$, it is always the case that $\{B\}$ [holds] for 5 time units.
Past MTL Formula	$\blacksquare(A \rightarrow \blacksquare_{[0,5]} B)$
Future MTL Formula	$\square(\Diamond_{[0,5]} A \rightarrow (B \mathcal{U} A))$

Table 5.2: MTL Formula Conversion Example [AGL+15; Ulu19a]

As can be seen in Table 5.2, the future-MTL formula of the first PSP is almost the same as the past-MTL formula of the second PSP, the sole difference being the usage of different temporal operators, i.e. future $always$ and past $always$. Additionally, when both PSPs are compared, it could be argued that they describe the same events, however, observed from different points. This is also depicted in Table 5.3. As implied in the table, by changing the future operators to their past equivalents and by reading the event traces in reverse, one could evaluate the future-MTL formulas by using only the past monitors. This is namely how the prototype handles future-MTL formulas. After a future-MTL formula is converted and its past variant is verified, its results are returned in the reversed manner in which the event traces were read, i.e. the first time point in the results is actually the last time point in the measurement data and the last time point in the results is the first time point in the measurement data. The reason for this is that the results correspond to the past verification of the formula and reversing the results and the event traces to their original future form would be wrong as the results would not correspond to it since they are originating from the past verification. Nevertheless, when the evaluation component verifies a future-MTL formula, an additional output field is included which indicates that the results and the event traces have been reversed.

	...	...	
	...	...	
$\square(A \rightarrow \square_{[0,5]}B)$	A	...	
$\downarrow$	...	B	
	...	B	
	...	B	
	...	B	$\uparrow$
	...	B	$\blacksquare(A \rightarrow \blacksquare_{[0,5]}B)$
	...	...	
	...	...	

Table 5.3: Future and Past Example Trace.

Plotting Component

A further component of the verification tools is the plotting component. The sole purpose of this component is to create plots containing the monitoring data, the verification results, and the interval information. These plots are created with the help of the matplotlib Python library [Mat]. An example of a plot created by the prototype is presented in Figure 5.3. The plots created by this component depict the measurement data used in the verification as lines. In the example plot in Figure 5.3, these are the blue and black lines. Additionally, the background color of the plot indicates what the verification result is in the given time intervals, for example from Figure 5.3 it can be concluded that the behavior specification was satisfied from the start of the experiment at time 0 until around the 300th time unit, after which the specification was violated.

Graphical User Interface

Taking into account the requirement given by some of the interviewed people that the tool should be also usable as a stand-alone tool, meaning without the need for Chaos Experiments and Chaos Toolkit, the prototype also provides a GUI. Since the verification tool already implements a REST API, it is used to also serve the user interface to the users. This interface can be used to easily create transient behavior specifications either by using MTL or PSPs, insert these into existing Chaos Experiments, and verify such specifications straight from the user interface without the need and the execution of Chaos Experiments. The GUI is created using HTML, CSS, JavaScript, the Bootstrap framework [Boo], and Jinja [Jin], which is Flask's template engine. The user interface is rather simple and minimalistic, and as a result, consists of only 7 web pages.

Figure 5.4 presents the welcoming page of the user interface. On this page, the user is presented with the three options that can be selected, namely the creation of a transient behavior specification, the insertion of such a specification into a Chaos Experiment, and the stand-alone verification of transient behavior specifications.

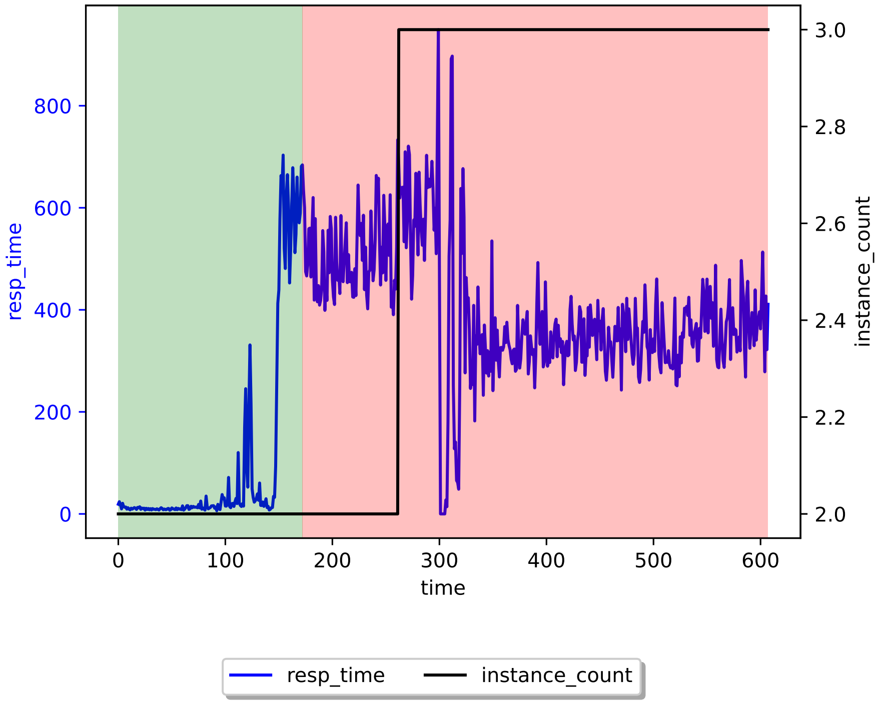


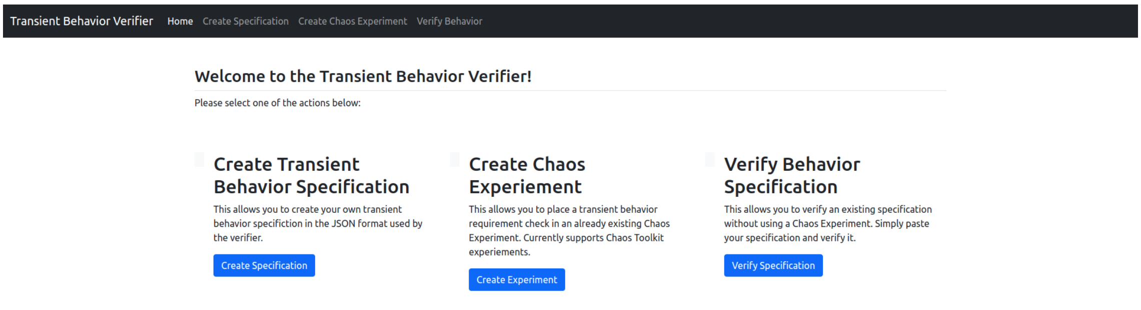
Figure 5.3: Result Plot Example.

Figure 5.5 presents the page allowing the user to create a behavior specification using an MTL formula. The page consists of numerous text fields that extract information about the formula, predicates, and measurements, which are then inserted into the specification JSON object and provided to the user as a downloadable file. Figure 5.6 presents the page that can be used to create transient behavior specifications with a PSP. This page contains almost the same fields as those for the MTL behavior specifications, the only difference being the fields related to the formalism used in the specification. Both the MTL and the PSP page allow the user to select the logic of the predicates used in the specification and the measurement source by using drop-down menus with pre-configured information. The page for the creation of the PSP behavior specifications also allows a selection of the supported PSPs by using a drop-down menu.

The page that allows the users to place a behavior specification already defined in the respective JSON format into a Chaos Experiment is presented in Figure 5.7. As shown in the figure, the page contains a simple text box where the JSON specification is to be pasted and a file input field for the Chaos Experiment.

Similarly, the page that allows users to verify behavior specifications directly from the user interface also contains a simple text box for the JSON specification and an optional file input field for the cases in which the verification uses a CSV file. This page is presented in Figure 5.8.

The page containing the results of the verification is shown in Figure 5.9. As shown in the figure, the page contains the Boolean result regarding the verification of the specification, the information regarding the intervals, and the resulting plot.



Transient Behavior Verifier Home Create Specification Create Chaos Experiment Verify Behavior

Welcome to the Transient Behavior Verifier!

Please select one of the actions below:

Create Transient Behavior Specification

This allows you to create your own transient behavior specifiction in the JSON format used by the verifier.

Create Specification

Create Chaos Experiement

This allows you to place a transient behavior requirement check in an already existing Chaos Experiment. Currently supports Chaos Toolkit experiements.

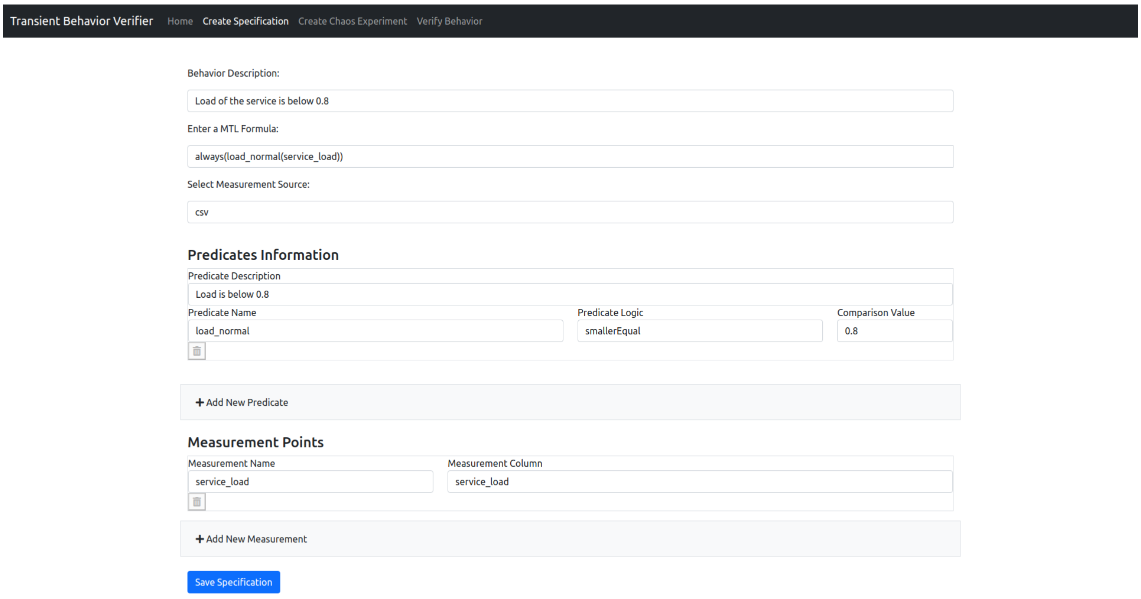
Create Experiment

Verify Behavior Specification

This allows you to verify an existing specification without using a Chaos Experiment. Simply paste your specification and verify it.

Verify Specification

Figure 5.4: Graphical User Interface - Home Page.



Transient Behavior Verifier Home Create Specification Create Chaos Experiment Verify Behavior

Behavior Description: Load of the service is below 0.8

Enter a MTL Formula: always(load_normal(service_load))

Select Measurement Source: csv

Predicates Information

Predicate Description: Load is below 0.8

Predicate Name	Predicate Logic	Comparison Value
load_normal	smallerEqual	0.8

+ Add New Predicate

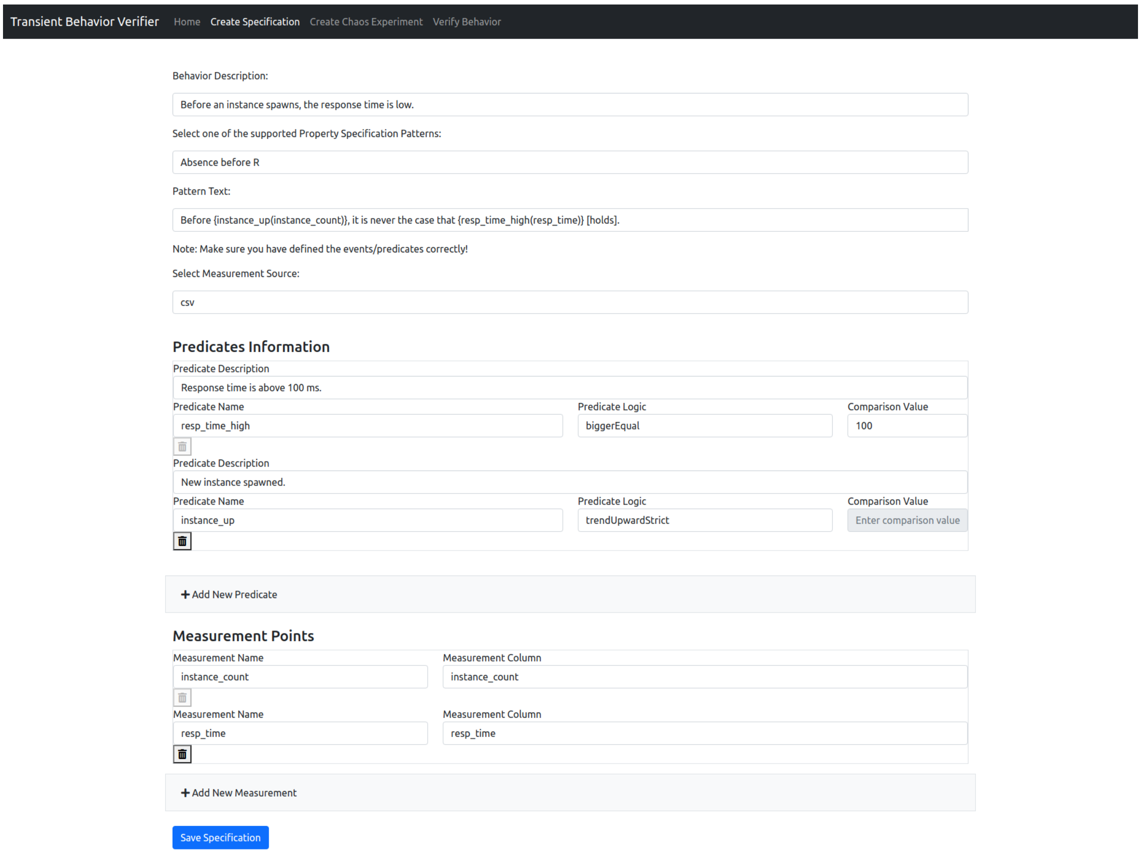
Measurement Points

Measurement Name	Measurement Column
service_load	service_load

+ Add New Measurement

Save Specification

Figure 5.5: Graphical User Interface - Behavior Specification Using MTL.



Behavior Description:

Before an instance spawns, the response time is low.

Select one of the supported Property Specification Patterns:

Absence before R

Pattern Text:

Before {instance_up(instance_count)}, it is never the case that {resp_time_high(resp_time)} [holds].

Note: Make sure you have defined the events/predicates correctly!

Select Measurement Source:

csv

Predicates Information

Predicate Description: Response time is above 100 ms.

Predicate Name	Predicate Logic	Comparison Value
resp_time_high	biggerEqual	100

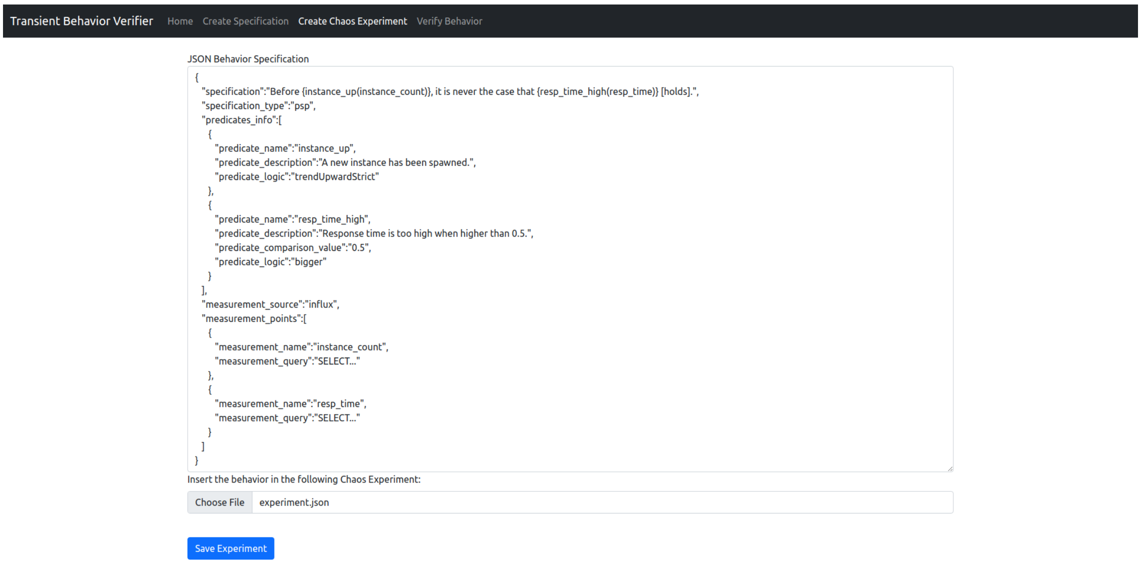
Predicate Description: New instance spawned.

Predicate Name	Predicate Logic	Comparison Value
instance_up	trendUpwardStrict	Enter comparison value

Measurement Points

Measurement Name	Measurement Column
instance_count	instance_count
resp_time	resp_time

Figure 5.6: Graphical User Interface - Behavior Specification Using PSPs.



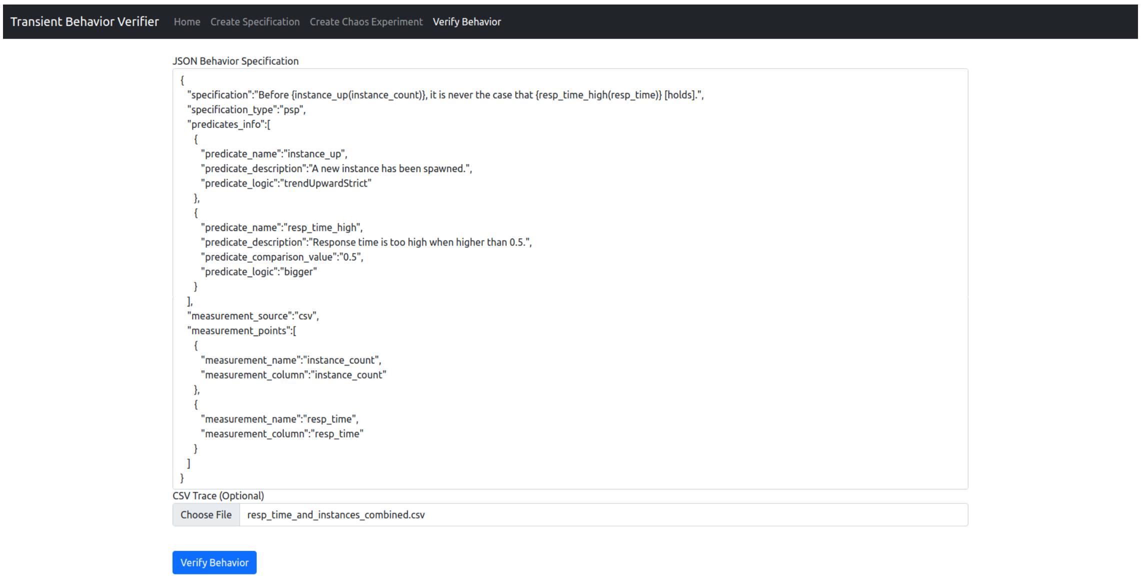
JSON Behavior Specification

```
{
  "specification":"Before {instance_up(instance_count)}, it is never the case that {resp_time_high(resp_time)} [holds].",
  "specification_type":"psp",
  "predicates_info":[
    {
      "predicate_name":"instance_up",
      "predicate_description":"A new instance has been spawned.",
      "predicate_logic":"trendUpwardStrict"
    },
    {
      "predicate_name":"resp_time_high",
      "predicate_description":"Response time is too high when higher than 0.5.",
      "predicate_comparison_value":"0.5",
      "predicate_logic":"bigger"
    }
  ],
  "measurement_source":"influx",
  "measurement_points":[
    {
      "measurement_name":"instance_count",
      "measurement_query":"SELECT..."
    },
    {
      "measurement_name":"resp_time",
      "measurement_query":"SELECT..."
    }
  ]
}
```

Insert the behavior in the following Chaos Experiment:

experiment.json

Figure 5.7: Graphical User Interface - Experiment Creation Page.



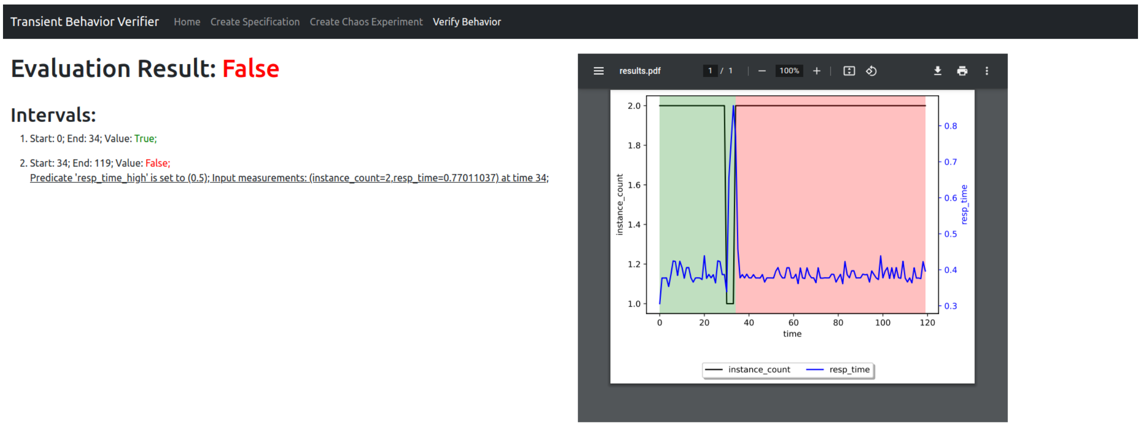
JSON Behavior Specification

```
{
  "specification":"Before {instance_up(instance_count)}, it is never the case that {resp_time_high(resp_time)} [holds].",
  "specification_type":"psp",
  "predicates_info":[
    {
      "predicate_name":"instance_up",
      "predicate_description":"A new instance has been spawned.",
      "predicate_logic":"trendUpwardStrict"
    },
    {
      "predicate_name":"resp_time_high",
      "predicate_description":"Response time is too high when higher than 0.5.",
      "predicate_comparison_value":"0.5",
      "predicate_logic":"bigger"
    }
  ],
  "measurement_source":"csv",
  "measurement_points":[
    {
      "measurement_name":"instance_count",
      "measurement_column":"instance_count"
    },
    {
      "measurement_name":"resp_time",
      "measurement_column":"resp_time"
    }
  ]
}
```

CSV Trace (Optional)

resp_time_and_instances_combined.csv

Figure 5.8: Graphical User Interface - Stand-alone Verification Page.



Evaluation Result: False

Intervals:

1. Start: 0; End: 34; Value: True;
2. Start: 34; End: 119; Value: False;
   Predicate 'resp_time_high' is set to (0.5); Input measurements: (instance_count=2,resp_time=0.77011037) at time 34;

Figure 5.9: Graphical User Interface - Verification Results Page.

6 Evaluation

In this chapter, the proposed extended Chaos Engineering approach is evaluated through the evaluation of its implementation. There are three different types of evaluations conducted in total, namely an evaluation regarding the correctness of the verification of transient behavior specifications, an evaluation of the applicability with Chaos Experiments and their execution, and an evaluation of the integration with other approaches, based on the compatibility with the simulation approach MiSim [Wag21]. The different types of evaluation also utilize the Goal Question Metric (GQM) approach [BCR]. The GQM approach is particularly useful to operationalize project goals into measurable metrics [BRZ]. The GQM models defined by the GQM approach contain namely conceptual goals, questions derived from these goals the answers to which should indicate whether the goals are met, and metrics suited to measure the answers to the questions [BCR]. Figure 6.1 depicts the structural hierarchy of the GQM models used in the following evaluations. For the purpose of the evaluation of the approach proposed by this thesis, three GQM models were defined, one for each type of evaluation. In the following sections, each evaluation is thoroughly described.

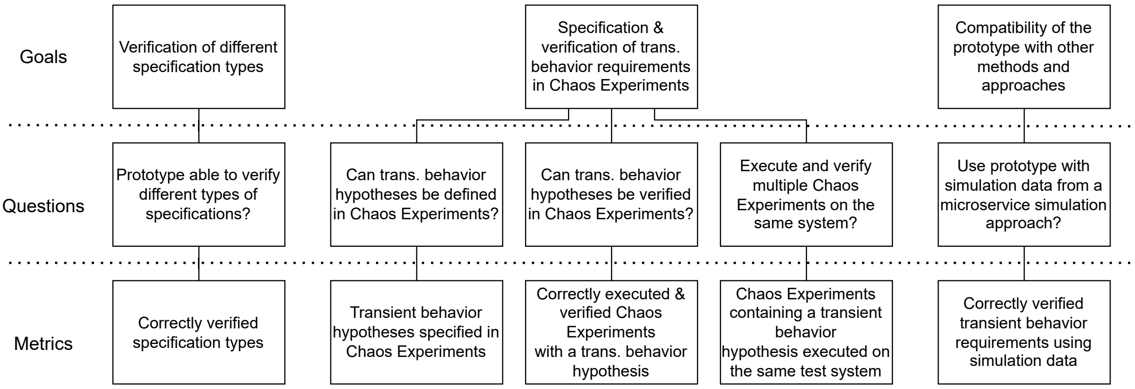

Goals
Verification of different specification types
Specification & verification of trans. behavior requirements in Chaos Experiments
Compatibility of the prototype with other methods and approaches
Questions
Prototype able to verify different types of specifications?
Can trans. behavior hypotheses be defined in Chaos Experiments?
Can trans. behavior hypotheses be verified in Chaos Experiments?
Execute and verify multiple Chaos Experiments on the same system?
Use prototype with simulation data from a microservice simulation approach?
Metrics
Correctly verified specification types
Transient behavior hypotheses specified in Chaos Experiments
Correctly executed & verified Chaos Experiments with a trans. behavior hypothesis
Chaos Experiments containing a transient behavior hypothesis executed on the same test system
Correctly verified transient behavior requirements using simulation data

Figure 6.1: GQM Models Overview.

6.1 Correctness Evaluation

The goal of this first evaluation is to examine whether the approach and respectively the prototype implementing it accomplish their goal to enable a verification of transient behavior requirements. From this goal, a question and a respective metric were also derived. The complete GQM model for this evaluation is presented Table 6.1.

Goal	Enable verification of different types of transient behavior requirements specified using Metric Temporal Logic and Property Specification Patterns against monitoring data of software systems.
Question	Is the prototype able to verify different types of transient behavior specifications?
Metric	Number of correctly verified transient behavior specification types.

Table 6.1: GQM Model for the Correctness Evaluation.

6.1.1 Method

To be able to answer the question defined by the GQM model and to successfully carry out the evaluation, several transient behavior specifications and respective event traces are required. To generate these, a benchmarking tool for MTL monitoring tools called Timescales [Ulu19a] is used. Even though Timescales is primarily focusing on the performance evaluation of monitoring tools, it can also be used to evaluate the correctness of the verification as it can generate the MTL formulas for 4 different PSPs with 4 different timing scopes and it is also able to generate event traces for each of the formulas as CSV files. The supported PSPs and scope combinations are the following:

1. Bounded Absence After Q
2. Bounded Absence Before R
3. Bounded Absence Between Q and R
4. Bounded Universality After Q
5. Bounded Universality Before R
6. Bounded Universality Between Q and R
7. Bounded Recurrence Globally
8. Bounded Recurrence Between Q and R
9. Bounded Response Globally
10. Bounded Response Between Q and R

Furthermore, the tool can generate two test suites, namely a small and a large suite. The small suite contains 10 MTL formulas, one for each supported PSP and scope combination, with short time bounds, in particular 10, and a trace length of ten thousand time units. The large suite contains 30 MTL formulas, 3 for each supported PSP and scope combination, with increasingly larger time bounds, from 10 up to 1000, and a trace length of a million time units. Furthermore, both suites can be configured to contain either past or future-MTL formulas. For the evaluation of the approach, both the small and large suites are generated two times, first containing past-MTL formulas, and then containing future-MTL formulas, together with the event traces for each behavior definition, i.e. past and future formula pair. By default Timescales generates traces in which the behaviors are satisfying the behavior definition, however, the tool allows to also generate traces that violate the behavior definition. As a result, both types of traces are generated for each suite. In total, for this evaluation, there are 80 MTL formulas, one half of them being in past-MTL and the other in future-MTL, with different timing constraints based on the 10 above-listed PSPs and scope combinations and 80 event traces in total, where for each formula pair there is a trace satisfying the behavior specification and another one that violates the specification. The MTL formulas and the respective event traces can be found in the supplementary material of this thesis [Zah22]. The evaluation is executed with the help of Postman [Pos], in which multiple collections containing the requests required for the evaluation are created. The collections containing the formulas and the traces that are satisfying the formulas are labeled as “Past Small Suite - True“, “Future Small Suite - True“, “Past Large Suite - True“ and “Future Large Suite - True“, whereas the collections in which the traces are violating the formulas are labeled as ‘Past Small Suite - False“, “Future Small Suite - False“, “Past Large Suite - False“ and “Future Large Suite - False“. Afterward, the evaluation is executed by executing each of the defined collections.

6.1.2 Results and Discussion

The results of the evaluation with each test suite are summarized in Table 6.2. As shown in the table, all of the 80 MTL formulas that are combinations of different PSPs and scopes contained in the test suites are correctly verified against 40 event traces that satisfy the respective behavior definitions, and another 40 event traces that violate them. As a result, the question in the GQM model from Table 6.1 is affirmatively answered. Thus, it is concluded that the goal to enable a verification of different types of transient behavior requirements is successfully achieved.

Test Suite	Number of initiated formula verifications	Successfully verified formulas
Past Small Suite - True	10	10
Past Small Suite - False	10	10
Future Small Suite - True	10	10
Future Small Suite - False	10	10
Past Large Suite - True	30	30
Past Large Suite - False	30	30
Future Large Suite - True	30	30
Future Large Suite - False	30	30

Table 6.2: Correctness Evaluation - Test Suite Results.

6.2 Chaos Experiment Evaluation

Another goal of the approach that is evaluated is the ability to specify and verify transient behavior requirements using MTL and PSPs as part of Chaos Experiments. Therefore, the questions of whether the transient behavior requirements can be defined in Chaos Experiments, verified as part of these, and whether multiple Chaos Experiments containing different transient behavior hypotheses can be verified using the same test system are examined. The full GQM model defined for this evaluation is presented in Table 6.3.

Goal	Enable the specification and verification of transient behavior requirements using MTL and PSP as part of Chaos Experiments.
Question 1	Can transient behavior hypotheses be defined as part of Chaos Experiments?
Question 2	Can transient behavior hypotheses be verified as part of the execution of Chaos Experiments?
Question 3	Can multiple Chaos Experiments containing different types of transient behavior hypotheses be executed and correctly verified on the same continuously running test system instance?
Metric 1	Number of transient behavior hypotheses specified as part of Chaos Experiments.
Metric 2	Number of correctly executed and verified Chaos Experiments containing a transient behavior hypothesis.
Metric 3	Number of Chaos Experiments containing a transient behavior hypothesis executed on the same test system instance.

Table 6.3: GQM Model for the Chaos Experiment Evaluation.

6.2.1 Test System

In order to be able to execute Chaos Experiments, first, a test system is required. The test system used during this evaluation is a mocked version of a payment calculation system [FHW+21] developed by an industrial partner of the university, who however should remain confidential, as a result of which, also some details about the system have to be kept hidden and obscured. This system is selected for the conduction of this evaluation as it has already been used in Chaos Engineering research [FHW+21], in which Chaos Experiments were successfully executed onto the system. An additional argument for the selection of this system is that by using an experimental setup similar to the one in the already-mentioned research [FHW+21], it is possible to measure certain metrics, such as response time, for the whole duration of the experiment, which would imply that the setup is able to measure and capture the exhibited transient behaviors. The mock of the system uses the microservice architecture style and consists of four services. The purpose of the first service called `API-Gateway` is to distribute the incoming load to two other services, respectively called `Service1` and `Service2`. Additionally, `Service1` is also able to forward calls to `Service2`. There is also an `Eureka` service used for service discovery. A component diagram of the mentioned components is presented in Figure 6.2. Furthermore, the test system includes a load generator capable of generating a specific number of requests for the endpoints exposed by the system. The information regarding the generated load is additionally saved into an InfluxDB database instance. In order to enable the execution of Chaos Experiments on the test system, the `API-Gateway`, `Eureka`, `Service1`, and `Service2` components were deployed on a local microk8s [Mic] Kubernetes installation. The Kubernetes deployment was configured in such a way that there were two instances of `Service1` and two instances of `Service2` hosted on different pods. In the following sections, three Chaos Experiments are described and executed on the presented system, after which their results are discussed.

	Chaos Experiment 1	Chaos Experiment 2	Chaos Experiment 3
Steady-state hypothesis	Deployment is fully available	Deployment is fully available	Deployment is fully available
Anomaly	One Service1 instance is killed	Sudden workload increase	Sudden workload increase
Transient behavior hypothesis type	MTL formula	PSP	MTL formula
Monitoring data source	InfluxDB	InfluxDB	Prometheus

Table 6.4: Executed Chaos Experiments - Overview.

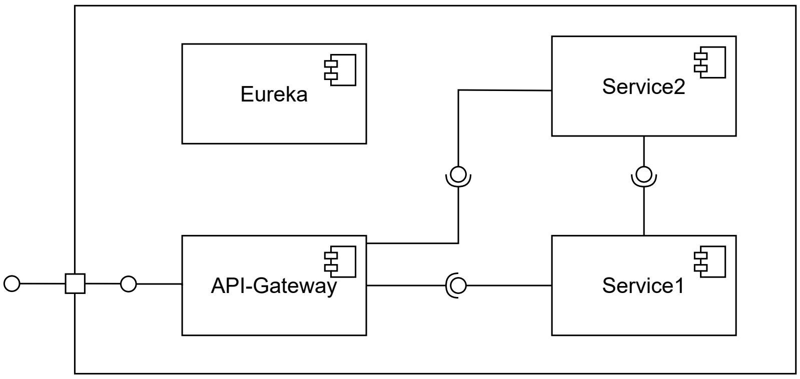


Figure 6.2: Test System - Architecture.

6.2.2 Method

In the following sections, three different Chaos Experiments are executed in order to provide answers to the questions defined in the GQM model regarding this evaluation. Two of the experiments include transient behavior hypotheses specified using MTL formulas and one using a PSP. Additionally, two of the experiments use the InfluxDB instance as the source of the measurement data, while the third one uses Prometheus as the source. A brief overview of the three Chaos Experiments and their characteristics is presented in Table 6.4.

Chaos Experiment With an MTL Transient Behavior Specification

First, an anomaly that is injected as part of this Chaos Experiment has to be selected. One common anomaly that can occur in such microservice systems as the test system described above and used in the experiment, is the failure of single components or services. Also, it is assumed that such types of failures are inevitable in the long run [New15]. Thus, a failure of a single instance is the selected anomaly that will be injected as part of this Chaos Experiment. The failure is defined by stopping one of the `Service1` pods while the system is in a normal mode of operation. The duration of the experiment is defined to be 120 seconds. The failure is introduced in the 30th second of the experiment. This is done in order to allow the system to adapt to the introduced workload and to get into a steady state in the first 30 seconds of the experiment. The steady-state hypothesis check is scheduled for the 120th second of the experiment as the system should recover until then and is checking whether the deployment is fully available. The hypothesis regarding the transient behaviors occurring in this experiment is informally defined as follows: Always when the instance count of a certain service drops from 2 to 1, a new instance has to be spawned in a maximum of 30 seconds and during this time the response time should be below 100 ms. The formal definition of the hypothesis specified using an MTL formula is as follows:

$$\blacksquare(\ (\ instance\_dead(instance\_count)\ \wedge\ resp\_time\_normal(resp\_time)\ )$$
$$\mathcal{S}_{[0,30]}\ instances\_normal(instance\_count)\ )$$

In order to give some meaning to the MTL formula, the logical predicates and the measurements have to be defined. For the purpose of this Chaos Experiment, the $instance\_normal$ predicate evaluates to $true$ when the $instance\_count$ values are equal to 2, the $instance\_dead$ predicate evaluates to $true$ when the $instance\_count$ values are equal to 1 and $resp\_time\_normal$ evaluates to $true$ when the $resp\_time$ values are lower than or equal to 100. For the purpose of the Chaos Experiment, the $instance\_count$ measurements are corresponding to the instance count of `Service1` and the $resp\_time$ measurements are corresponding to the response time of the requests sent to the `API-Gateway` by the load generator, and both of the measurements are preserved in the InfluxDB instance. The above-described Chaos Experiment expressed in Chaos Toolkit's format can be found in Listings A.1 and A.2 as well as in the supplementary material of the thesis [Zah22].

The above-defined Chaos Experiment was executed on the test system while the load generator was generating 20 requests per second to the `API-Gateway`, after which they are distributed to the remaining services. The plot created by the transient behavior verification tool visualizing the results of transient behavior verification, the instance count of `Service1`, and the response time of the test system is presented in Figure 6.3. In the plot, the red and blue lines present the measurement data regarding the response time and the instance count, whereas the green and red background colors indicate the different verification results at the different time intervals during the experiment. For example, as can be seen in the plot, the behavior specification is evaluated as $true$ until about the 35th second, which is indicated by the green background color, and then the evaluation turns to $false$, which is also indicated by the red background color. A deeper look into the measurement data which is also presented in the plot reveals that the behavior specification was indeed violated around the 35th second, in which the instance count of `Service1` was 1 and the response time of the system under test was around 250 ms. As the transient behavior specification requires the response time to be at most 100 ms., the $false$ verification result indicated by the red color starting from around the 35th second is indeed correct. In conclusion, it can be stated that the above described transient behavior specification including an MTL formula can be successfully incorporated into a Chaos Experiment and is correctly verified during its execution, meaning that the violation of the specification is correctly identified.

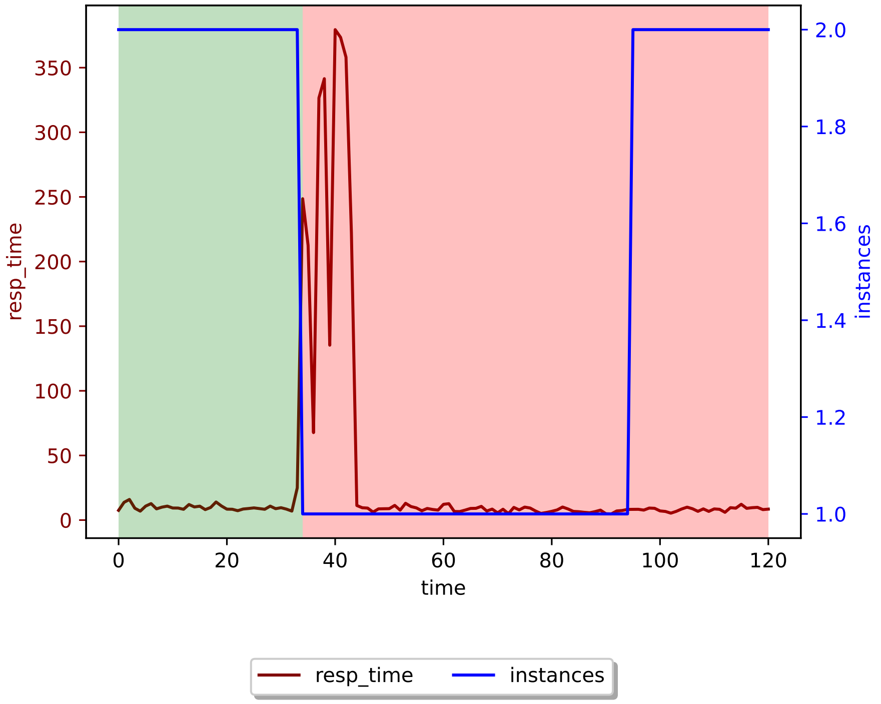


Figure 6.3: Chaos Experiment 1 - Verification Results.

Chaos Experiment With a PSP Transient Behavior Specification

The anomaly that is injected in the second Chaos Experiment is a significant increase in the incoming workload to the `API-Gateway`. This is selected in order to examine the autoscaling capabilities of the test system and since a sudden increase in the workload has also been identified as a possible cause of transient behaviors [Bec20]. The duration of this experiment is 600 seconds which is 10 minutes. At the start of the experiment, the incoming load to the `API-Gateway` component is one request per second. Until the end of the experiment, the workload reaches 1000 requests per second. The steady-state hypothesis of this second experiment is checked at the 600th second and is the same as in the first one, namely that the deployment is fully available. The transient behavior hypothesis in this experiment can be informally defined as follows: When the response time of the system exceeds 150 ms. then a new instance of `Service1` must be spawned in the next 30 to 60 seconds. The formal definition of the transient behavior hypothesis using a PSP is the following:

> Globally, if $\{resp\_time\_high(resp\_time)\}$ [has occurred] then in response $\{instance\_increase(instance\_count)\}$ [eventually holds] between 30 and 60 time units.

Additionally, the logical variables in the above-defined PSP have to be specified in order to give the PSP definition its actual meaning. The $resp\_time\_high$ predicate is $true$ when the $resp\_time$ values are bigger than or equal to 150, which in this case corresponds to milliseconds. The $instance\_increase$ predicate evaluates to $true$ when the $instance\_count$ values concerning `Service1` are bigger than 2. Similarly to the previous experiment, the measurement values are preserved in the InfluxDB instance. As next, the individual parts of the Chaos Experiment have to be put together and defined in Chaos Toolkit's syntax. Since the increase of the load is configured in and initiated from the load generator component, the method section in the experiment is not required since Chaos Toolkit does not have to interact with the load generator. However, since the method field is required by Chaos Toolkit, a simple probe is specified in the method so that the lack of a method does not have any effect on the execution of the experiment. The Chaos Experiment defined in Chaos Toolkit's format can be found in Listings A.3 and A.4, and also in the supplementary material of the thesis [Zah22]. When the experiment was executed, the transient behavior verification tool generated the plot presented in Figure 6.4. In the plot, the red line is depicting the response time of the system and the blue line depicts the instance count of `Service1`. Also in this plot, the green and red background colors indicate the verification result with regard to the different time frames during the experiment.

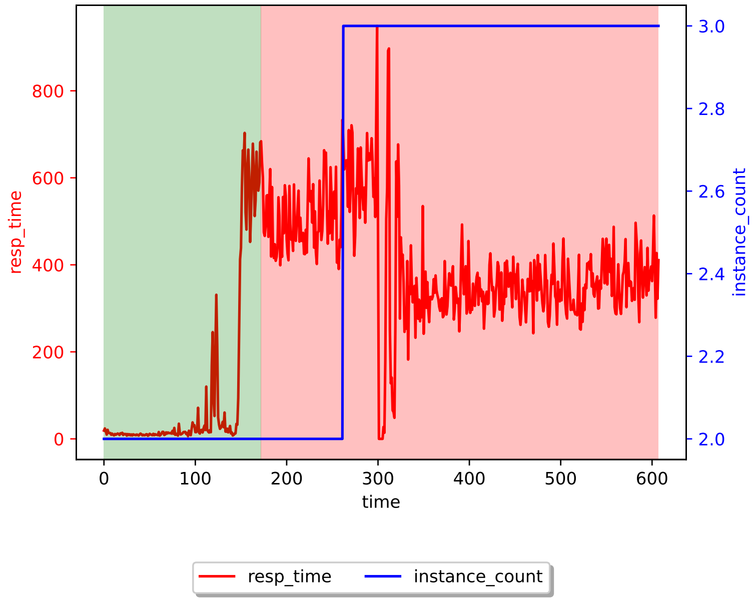


Figure 6.4: Chaos Experiment 2 - Verification Results.

As depicted in the plot above, the response time of the system exceeds 150 ms around the 110th second and the new instance is spawned around the 260th second, which is exceeding the constraint in the behavior specification by around 90 seconds. Therefore, the verification results indicating that the transient behavior requirement was violated are correct and the verification tool is capable of successfully verifying such behavior requirements specified as PSPs.

Chaos Experiment Evaluating the Integration of Prometheus as a Monitoring Tool

The purpose of this Chaos Experiment is to evaluate the integration of Prometheus as a source of monitoring data since one of the interviewees requested its integration and the fact that Prometheus is an industry-standard with regard to monitoring approaches and tools. The anomaly that is introduced in this last Chaos Experiment is the same as in the previous one, namely an increase in the incoming workload. The steady-state hypothesis of the experiment is also the same, namely that the deployment is fully available. The only part differing in this experiment is the transient behavior requirement which requires the CPU usage of both `Service1` pods should be below 0.8 cores. The same requirement is formally specified as the following MTL formula:

$$\blacksquare(\ load\_normal(service1\_1\_load)\ \wedge\ load\_normal(service1\_2\_load)\ )$$

To complete the definition of the behavior, the logical predicates in the formula are defined as follows. The $load\_normal$ predicate is $true$ when the values it receives are below 0.8, and the $service1\_1\_load$ and $service1\_2\_load$ measurements are regarding the number of cores used by each `Service1` pod. The Chaos Experiment defined in Chaos Toolkit's format can be found in Listings A.5 and A.6 and also in the supplementary material of the thesis [Zah22]. The plots generated from the transient behavior verification tool after executing the experiment are presented in Figure 6.5. The red and blue lines are depicting the utilization of the individual pods and similarly to the previous plots, the green and red background colors are indicating the verification results at the different time points.

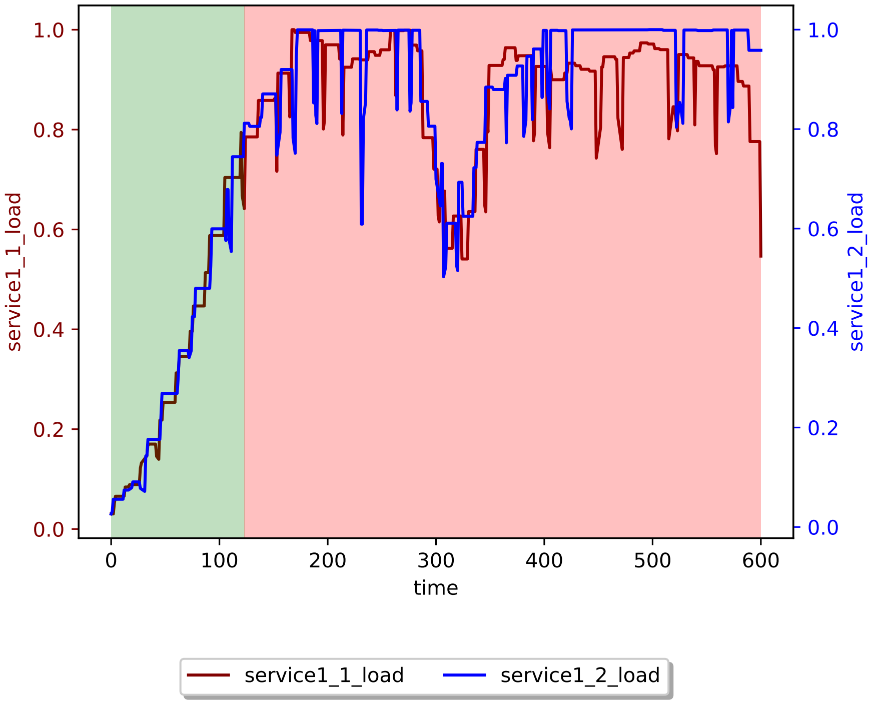


Figure 6.5: Chaos Experiment 3 - Verification Results.

As presented in the plot above, the utilization of both pods exceeds 0.8 cores at around the 110th second. As a result, it is concluded that the verification tool correctly verified the violation of the behavior specification, while also using Prometheus as the source of the monitoring data.

6.2.3 Results and Discussion

In the sections above, three Chaos Experiments were successfully specified and executed using the extended Chaos Engineering approach. Since each of these included a transient behavior hypothesis specified using either an MTL formula or a PSP, the first question of the GQM model in Table 6.3

is positively answered. Furthermore, all of the transient behavior hypotheses were successfully and correctly verified as part of the execution of the experiments. Therefore, also the second question from the GQM model is answered positively. Lastly, all of the specified Chaos Experiments which contained different transient behavior hypotheses using different types of formalisms, i.e. MTL formulas and PSPs, were executed on the same test system and were correctly verified. Thus, also the third question in the GQM model is positively answered. As a result, the goal to enable the specification and verification of transient behavior requirements as part of Chaos Experiments has been achieved.

6.3 Evaluation of the Compatibility With Other Approaches

As elicited in the expert interviews and included in the requirements for the extended Chaos Engineering approach, it must also be possible to use the approach as a stand-alone analysis without the need to run Chaos Experiments. Thus, a further goal of the approach proposed in this thesis is to be compatible with other approaches, by offering the verification of transient behavior requirements to these. One approach that can examine the resilience of software applications similarly to Chaos Engineering, however, by simulating experiments instead of actually running such, is MiSim [Wag21]. One important property of the usage of simulations is that they are particularly useful during design time, where the resilience of a system can be simulated and analyzed before the system is built. In comparison, the Chaos Engineering approach requires the existence of the analyzed systems. Introducing the ability to verify transient behavior requirements to the MiSim approach would further improve the resilience analysis that the tool is able to provide. Therefore, the ability to provide the verification of transient behavior requirements to further approaches such as MiSim is defined as an additional goal of the proposed approach and this goal is also used to derive additional questions and metrics which are summarized in the GQM model presented in Table 6.5.

6.3.1 Method

In order to evaluate the compatibility with MiSim, first, the architecture on which Chaos Experiments will be simulated must be defined. The architecture that is defined as part of this evaluation is graphically presented in Figure 6.6. The architecture contains three components, namely an `API-Gateway`, `Service1`, and `Service2`. The `API-Gateway` is the entry point of the application, which routes the incoming requests to `Service1`, of which there are two instances that route their incoming requests to `Service2`. The simulated architecture is intentionally similar to the architecture of the test system used in Section 6.2 albeit with minor differences. The main differences are in that there is only a single instance of `Service2` and that the `API-Gateway` cannot directly send requests to `Service2`. In addition, the MiSim architecture model does not include some of the properties of the test system from Section 6.2, such as its resilience mechanisms, but it does include an autoscaler for `Service1` to simulate Kubernetes' spawning of pods. The usage of an architecture model similar to the existing architecture of the test systems allows a comparison between the simulated data and the real measurements and a comparison of the results. The architecture model defined using MiSim's JSON format can be found in Listings A.7 and A.8 as well as in the supplementary materials [Zah22].

After the architecture is defined, the next step is to define an experiment that is simulated. The experiment is specified to have a duration of 120 seconds and at the 30th second, one of the `Service1` instances is killed. The experiment is also intentionally similar to the one executed in the Chaos Experiment evaluation in Section 6.2 in order to be able to detect any anomalies in the results. The experiments in MiSim are also defined as JSON objects and the above-described experiment is shown in Listing A.9. When the experiment gets simulated, the resulting simulation data is written into CSV tables, some of which are also provided in the supplementary materials of the thesis [Zah22]. Additionally, MiSim can create plots visualizing the data from the tables. Figure 6.7 presents the simulated response time of the example architecture whereas Figure 6.8 presents the instance counts throughout the experiment.

Goal	Allow compatibility of the prototype with other methods and approaches that can capture transient behaviors in a discrete-time manner and offer the ability to specify and verify transient behavior requirements to these.
Question	Can the prototype be used with simulation data originating from approaches capable of simulating microservice architectures and their resilience?
Metric	Number of correctly verified transient behavior requirements using simulation data.

Table 6.5: GQM Model for the Evaluation of the Compatibility With Other Approaches.

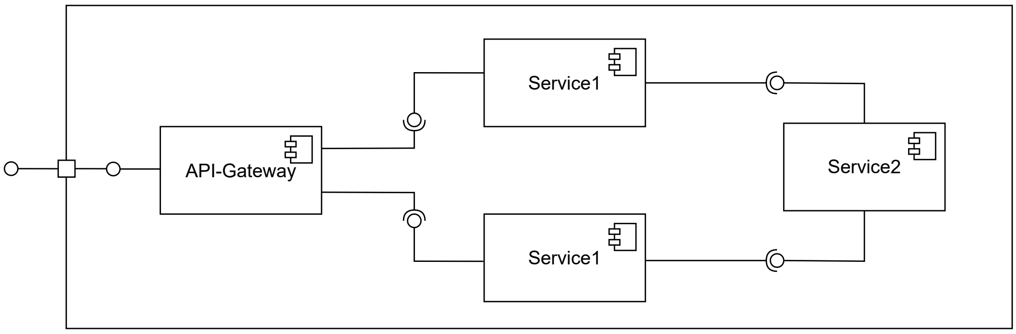


Figure 6.6: MiSim Simulated Architecture - Component Diagram.

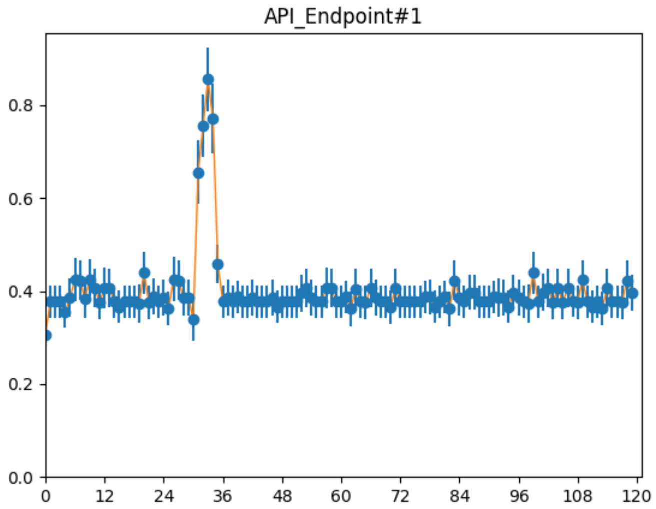


Figure 6.7: Simulated Response Time.

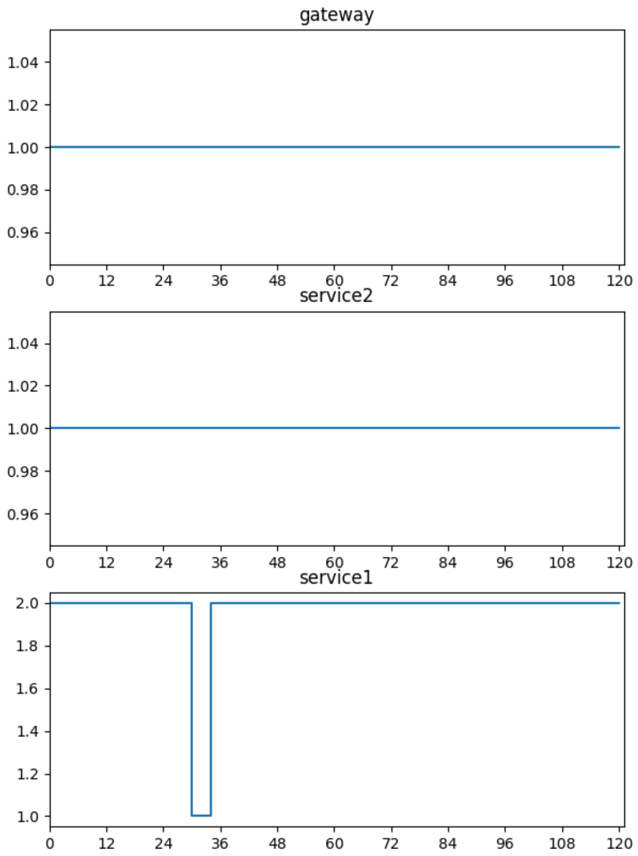


Figure 6.8: Simulated Instance Counts.

Next, the two transient behavior requirements that are verified against the simulation data are defined. The first transient behavior requirement is informally defined as follows: Whenever the number of instances of `Service1` drops from 2 to 1, it should rise back to 2 within a maximum of 3 seconds. This requirement can be formally defined using an MTL formula as follows:

$$\blacksquare(\ instance\_dead(instance\_count)\ \mathcal{S}_{[0,3]}\ instances\_alive(instance\_count)\ )$$

The *instance_dead* predicate is evaluating to *true* when the *instance_count* measurement is equal to 1 and the predicate *instance_alive* is *true* when *instance_count* is equal to 2. Additionally, the *instance_count* measurement is referring to the number of instances of `Service1`.

Since MiSim outputs the simulation results as CSV files, the stand-alone analysis functionality of the verification tool provided by the GUI is used to verify the above-defined behavior requirement. Furthermore, for the purpose of this behavior verification, the output table regarding the instance counts of the services has to be adjusted slightly. The original table produced by MiSim contains information only about the moments when there are changes in the number of instances. For example, the table starts with timestamp 0 and instance count of 2 and then jumps straight to timestamp 30 and instance count of 1. Since the prototype requires measurements for each time unit, the data is manually adjusted to include the times in between the changes. The JSON object that contains the transient behavior specification and that is verified using the verification tool is presented in Listing A.10. The resulting plot created by the verification tool is presented in Figure 6.9. As shown in Figure 6.9, the time interval of 3 time units specified in the transient behavior requirement is violated, since the service needs 5 time units to respawn as depicted in Figure 6.8. As a result, it can be concluded that the first transient behavior requirement is successfully and correctly verified using the simulation data.

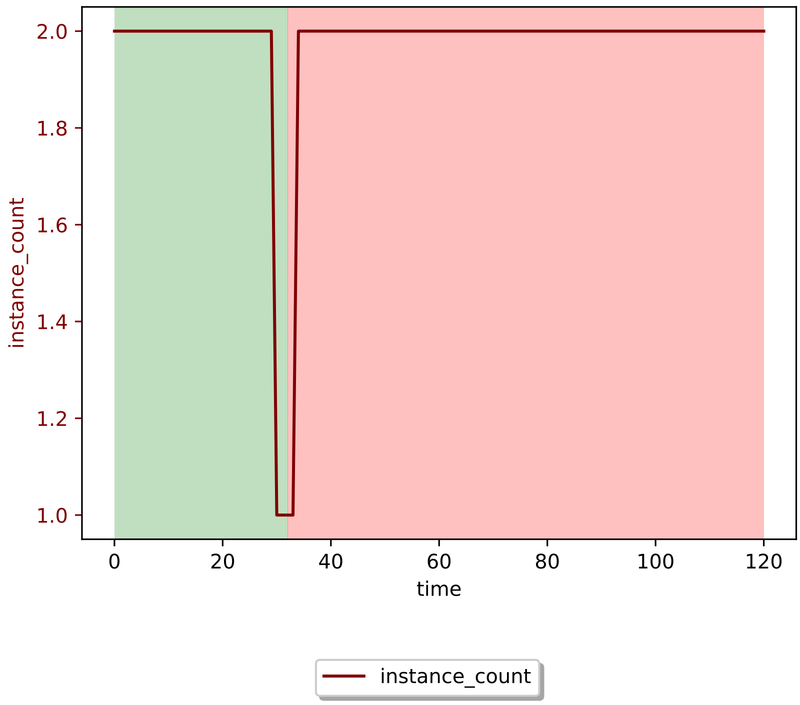


Figure 6.9: Compatibility Evaluation - Verification Result Plot 1.

The second transient behavior requirement that is verified using the simulation data produced by MiSim can be informally defined as follows: Before the second instance of `Service` is respawned, the response time of the systems should be below 0.5. Formally defined using a PSP the requirement is the following:

> Before $instance\_up(instance\_count)$, it is never the case that $resp\_time\_high(resp\_time)$ [holds].

The $instance\_up$ predicate in the behavior specification is defined as a strictly increasing function, i.e. it is evaluating to $true$ when the $instance\_count$ measurement values increase from 1 to 2. The $resp\_time\_high$ predicate is evaluating to $true$ when the $resp\_time$ measurement for the response time of the system is higher than 0.5. The transient behavior requirement expressed as a JSON object is presented in Listing A.11. Similarly to the previous behavior specification, also this one has to be verified by using the GUI of the verification tool. Furthermore, the output CSV tables of MiSim had to be adjusted once again, since the information regarding the instance counts and the response times is located in two separate tables. As a result, the two tables had to be combined into a new one. After the verification, the resulting plot presented in 6.10 was created. Looking at the data in Figures 6.8, 6.7, and 6.10, it is concluded that the verification component was able to successfully verify also the second transient behavior requirement against the simulation data produced by MiSim.

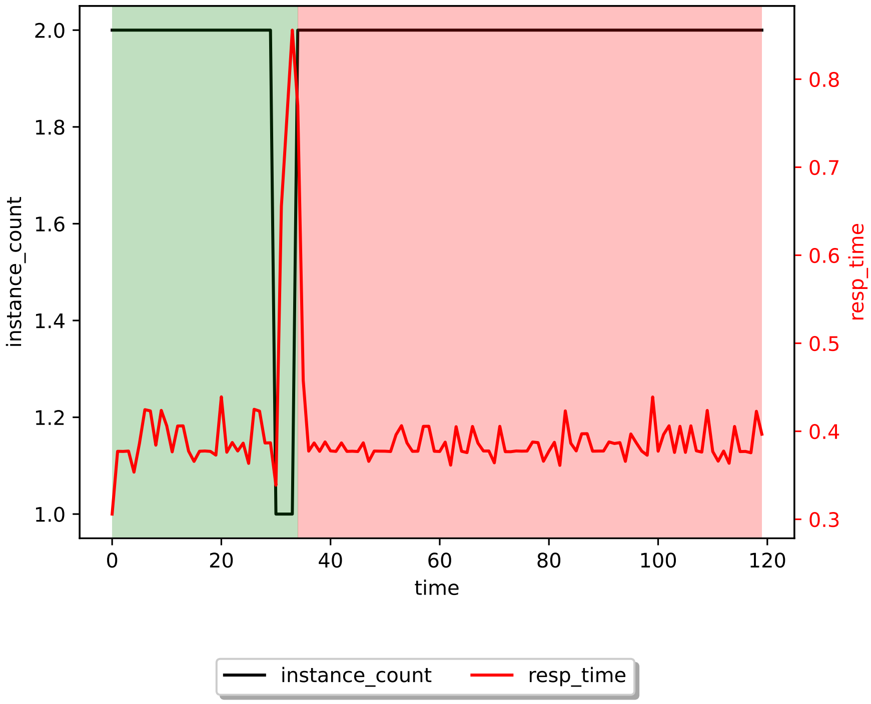


Figure 6.10: Compatibility Evaluation - Verification Result Plot 2.

6.3.2 Results and Discussion

In this evaluation, two transient behavior hypotheses are defined using both an MTL formula and a PSP. All of these hypotheses are then verified against simulation data produced by MiSim. Each of the hypotheses is also correctly verified against the simulation data required for its validation. As a result, the question from the GQM model in Table 6.5, whether the proposed approach can be used with simulation data, is positively answered. Thus, it can be concluded that the goal to allow compatibility with other approaches and to offer the ability to verify transient behavior hypotheses to these is achieved.

6.4 Evaluation of Research Questions

In this section, the results of the thesis are discussed with regard to the research questions formulated in Chapter 1, presented once again below.

RQ1 How can a transient behavior requirement be specified in the context of Chaos Engineering and Chaos Experiments?

RQ2 How can the Chaos Experimentation process incorporate a verification of transient behavior requirement specifications?

RQ3 What does a concept for an extended Chaos Engineering approach contain and how can it be implemented into a prototype?

RQ4 Is the extended Chaos Engineering approach able to correctly verify different transient behavior requirements?

RQ5 Is the extended Chaos Engineering approach capable of correctly executing Chaos Experiments and verifying transient behavior requirements as part of the experiments?

RQ6 Is the extended Chaos Engineering approach compatible with other approaches and tools?

Regarding the first research question, the results of the evaluation show that MTL and the PSPs are two formalisms capable of describing transient behavior requirements as part of the Chaos Engineering approach. This is further solidified by the multiple successfully executed Chaos Experiments containing namely MTL and PSP transient behavior specifications. Concerning the second research question, the concept for the extended Chaos Engineering approach proposes the integration of runtime verification as an additional step in the Chaos Experimentation process. The evaluation of the approach and its implementation shows that this is a viable extension to the experimentation process. With regard to the third research question, the thesis proposes a concept for an extended Chaos Engineering approach in Chapter 4, consisting of a transient behavior hypothesis, a transient behavior verification, a result visualization, a GUI, and a transient behavior monitoring approach. Furthermore, the concept is successfully implemented into a prototype, which is described in Chapter 5. After conducting the correctness evaluation in Section 6.1 and showing that the approach and its implementation are capable of verifying various different transient behavior specifications, the fourth research question is answered positively. Additionally, the Chaos Experiment evaluation carried out in Section 6.2, particularly the three Chaos Experiments containing transient behavior hypotheses which are correctly executed and verified by the extended approach, demonstrate that the approach is able to execute and verify Chaos Experiments containing such hypotheses. Thus, the fifth research question is also answered positively. Moreover, the evaluation conducted in Section 6.3 shows that the extended approach and its implementation are able to provide the verification of transient behavior requirements to other tools and approaches. Therefore, the last research question is also answered positively.

6.5 Threats to Validity

One potential threat to the validity of the research and in particular to the evaluation and the results might be the selection of the experimental setup used to run Chaos Experiments during the evaluation. In particular, the test system and its behaviors have been already thoroughly studied by other research projects [FHW+21]. This previous knowledge about the test system might have made the specification of the transient behavior hypotheses and the verification of these easier than the same with a different system, which was not previously studied. Thus, the results and findings gathered with the selected test system might not be applicable or transferable to other systems. To mitigate this threat, the selection of the test system was discussed with the supervisors of the thesis. Nevertheless, the microservice architecture style and the technologies utilized by the test system and also by an increasing number of modern systems would imply that the results and the exhibited behaviors would also be observable with other systems utilizing the same properties.

Additionally, the selection of the experts that were interviewed in order to elicit requirements regarding the extended Chaos Engineering approach could be a further threat to the validity of the results of the thesis. A different selection of interviewees with different experiences and backgrounds could have resulted in different answers to the questions and as a result, different requirements for the approach. To mitigate this threat, the selection pool and the inclusion of candidates in it were thoroughly discussed with the supervisors of the thesis. However, the number of interviewed people could also be a threat, since the selection of experts was rather limited, and interviewing additional people could have resulted in different answers and requirements.

Another potential threat to the validity of the thesis and its results are the expert interviews and in particular, the questions that the interviewees were asked. It might have been the case that the questions were not selected correctly in order to elicit the necessary and correct requirements for the extended Chaos Engineering approach. As a result, a different method for conducting the expert interviews and a different selection of questions might have resulted in a different extension to the Chaos Engineering approach. Furthermore, the questions asked during the interviews might have been biased by the author of the thesis. To mitigate these, the interview questions were reviewed multiple times by the supervisors of the thesis.

6.6 Limitations

One limitation of the implementation of the approach is that the transient behavior hypothesis has to be specified in the probes of the steady-state hypothesis. This is a limitation since the hypothesis regarding the transient behavior is conceptually different from the steady-state hypothesis. Since these two hypotheses are inherently different, the transient behavior hypothesis should not be a part of the steady-state hypothesis. This limitation is originating from the Chaos Engineering tool selected as part of the prototype, namely Chaos Toolkit. In Chaos Toolkit, only the tolerances of the probes in the steady-state hypothesis are checked and all tolerances that are specified outside these do not get checked. As a result, in order for the tolerances of the transient behavior hypothesis to be checked, this hypothesis has to be included in the probes of the steady-state hypothesis. If at a later point in time a check of the tolerances specified in the method is introduced, the transient behavior hypothesis could be easily moved to a probe in the method.

Another limitation of the implementation of the approach is the visualization of the verification results of future-MTL formulas. Since the future temporal operators have to be converted into their past equivalents, the event trace has to be read in the reverse direction and the results are also gathered in a reverse manner. As a result of this, the visualization of the results and the intervals in it are also reversed in order to comply with the past variant of the formula. Since this reversal of all of the results is contradicting the original direction of the event traces, the results of the verification of future-MTL formulas might be confusing for the users of the approach and are therefore counted as a limitation of the approach. Nevertheless, the final result of the verification is unaffected by this limitation.

A further limitation of the prototype is that there is no consistent way to define the beginning and the end of the event traces that should be used in the verification. For example, when InfluxDB is used as the monitoring tool, the beginning and the end of the trace have to be manually specified as part of the query string. However, when Prometheus is used as the measurement data source, the definition of the measurement points in the behavior specification allows the definition of the start and end times of the event trace that is used during the verification in separate fields. The definition of those however still has to be done manually. On the contrary, when a CSV file is used as the monitoring data source, the definition of a start and end point of the trace is not supported and the whole CSV file is used.

7 Conclusion and Future Work

This chapter concludes the thesis by providing a summary of the conducted research and discusses possible future work.

7.1 Summary

This work examines how the traditional Chaos Engineering approach and in particular the Chaos Experimentation process can be extended in order to enable the specification of requirements regarding the transient behaviors of software systems and to verify them as part of Chaos Experiments. To achieve this, first formalisms, namely temporal logic and the PSPs, capable of expressing transient behaviors and their temporal properties, are researched. In addition, the Chaos Engineering approach and its tools are also researched. After some initial ideas are gathered, they are implemented into an initial prototype with the purpose of presenting this initial prototype to people with Chaos Engineering expertise with the goal of studying the feasibility of the prototype and the concepts included in it and also gathering their requirements for an extended Chaos Engineering approach, that includes the verification of transient behavior requirements. After conducting multiple interviews and eliciting requirements for the extended approach, a concept for the approach is proposed. The concept contains various new elements, namely a transient behavior hypothesis specification, a transient behavior verification, a result visualization, a graphical user interface, and a transient behavior monitoring approach. Afterward, a prototype implementing the concept is developed with the goal of evaluating the extended Chaos Engineering approach. The approach is evaluated in three different ways, first, its correctness is evaluated using benchmark data, then a series of Chaos Experiments containing transient behavior hypotheses are executed using the approach, and afterward, the compatibility with other approaches and tools is evaluated. All of the above-listed evaluations are successfully completed, proving that the extended approach fulfills its goals and purpose. Additionally, the research questions formulated at the beginning of the thesis are addressed and answered. Lastly, the threats to the validity of the research and the limitations of the proposed approach are discussed.

7.2 Future Work

One aspect that can be further studied is the optimization of transient behavior specifications. The approach presented in this thesis showed that it is capable of verifying whether the specifications hold or are violated, however, an approach for optimizing the specification based on the verification results would be particularly useful. For example, assuming there is a service that actually takes 15 seconds to be revived, after a specification that requires the service to be revived in 30 seconds is verified, the optimization approach would be able to adjust and update the specification to reflect the actual time the service takes to be revived, i.e. update the specification from 30 seconds to 15 seconds.

Another aspect that can be studied is the introduction of further runtime verification monitors for other temporal logics or even other types of formalisms. Additionally, the existing catalog of PSPs that is supported by the prototype can be further expanded, and if additional temporal logics are introduced into the specification, also the mappings of the PSPs into these newly introduced temporal logics can be added.

In addition, further monitoring tools and approaches can also be studied. Since at the time of writing the approach only supports discrete-time event traces, it would be beneficial to examine how other types of event traces with different time interpretations such as continuous can be included in the extended Chaos Engineering approach. Additionally, the introduction of additional discrete-time monitoring tools is also a thing that can be considered.

Moreover, the limitation of Chaos Toolkit which is part of the prototype, which requires the specification of the transient behavior hypothesis as part of the probes of the steady-state hypothesis can be addressed. Research can be conducted in order to find possible ways to extend the Chaos Experiments of Chaos Toolkit in order to enable the specification of the transient behavior hypothesis outside the probes of the steady-state hypothesis. This may include the addition of different types of hypotheses or the introduction of a tolerance check for probes outside the steady-state hypothesis.

An additional aspect that can be further studied is the usability and the user experience of the approach and its prototype implementation introduced in this thesis. A user study with Chaos Engineering practitioners as subjects, evaluating the usability of the prototype and how practitioners would use it to run Chaos Experiments, could also provide further insights into how the tool can be extended to better support the needs of the practitioners.

Bibliography

[AGL+15] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, A. Tang. “Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar”. In: *IEEE Transactions on Software Engineering* 41.7 (2015), pp. 620–638. DOI: 10.1109/TSE.2015.2398877 (cit. on pp. 1, 10–12, 27, 32, 35, 41).

[AH92] R. Alur, T. A. Henzinger. “Logics and models of real time: A survey”. In: *Real-Time: Theory in Practice*. Ed. by J. W. de Bakker, C. Huizing, W. P. de Roever, G. Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 74–106. ISBN: 978-3-540-47218-6 (cit. on pp. 8, 10).

[BCE+03] M. Bruneau, S. E. Chang, R. T. Eguchi, G. C. Lee, T. D. O’Rourke, A. M. Reinhorn, M. Shinozuka, K. J. Tierney, W. A. Wallace, D. von Winterfeldt. “A Framework to Quantitatively Assess and Enhance the Seismic Resilience of Communities”. In: *Earthquake Spectra* 19 (2003), pp. 733–752 (cit. on p. 14).

[BCR] V. R. Basili, G. Caldiera, H. D. Rombach. *The Goal Question Metric Approach* (cit. on p. 47).

[Bec20] S. Beck. “Evaluating human-computer interfaces for specification and comprehension of transient behavior in microservice-based software systems”. Master’s Thesis. University of Stuttgart, 2020. DOI: http://dx.doi.org/10.18419/opus-11846 (cit. on pp. 2, 8, 53).

[BFH+21] S. Beck, S. Frank, M. A. Hakamian, L. Merino, A. van Hoorn. “TransVis: Using Visualizations and Chatbots for Supporting Transient Behavior in Microservice Systems”. In: *Proceedings of the 9th IEEE Working Conference on Software Visualization (VISSOFT 2021)*. Accepted. 2021 (cit. on p. 14).

[BK08] C. Baier, J.-P. Katoen. *Principles of model checking*. The MIT Press. London, England: MIT Press, Apr. 2008 (cit. on p. 10).

[BLS07] A. Bauer, M. Leucker, C. Schallhart. *Runtime Verification for LTL and TLTL*. Tech. rep. 2007 (cit. on p. 13).

[Boo] Bootstrap. *Official Website*. https://getbootstrap.com/. (Accessed on 07/28/2022) (cit. on p. 42).

[BRZ] B. Boehm, H. D. Rombach, M. V. Zelkowitz. *Foundations of Empirical Software Engineering* (cit. on p. 47).

[CCR13] I. M. Copi, C. Cohen, V. Rodych. *Introduction to Logic*. en. 14th ed. Pearson custom library. London, England: Pearson Education, June 2013 (cit. on p. 12).

[Chaa] Chaos Mesh. https://chaos-mesh.org/. (Accessed on 07/28/2022) (cit. on p. 31).

[Chab] Chaos Monkey. *Chaos Monkey - Documentation*. https://netflix.github.io/chaosmonkey/. (Accessed on 07/28/2022) (cit. on p. 31).

[Chac] Chaos Toolkit. *Official Website*. https://chaostoolkit.org/. (Accessed on 07/28/2022) (cit. on pp. 8, 9, 14, 31).

[Chad] ChaosBlade. https://chaosblade.io/. (Accessed on 07/28/2022) (cit. on p. 31).

[CL12] J. Cámara, R. de Lemos. "Evaluation of resilience in self-adaptive systems using probabilistic model-checking". In: *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2012, pp. 53–62. DOI: 10.1109/SEAMS.2012.6224391 (cit. on pp. 1, 13).

[CZ18] C. Czepa, U. Zdun. "On the Understandability of Temporal Properties Formalized in Linear Temporal Logic, Property Specification Patterns and Event Processing Language". In: *IEEE Transactions on Software Engineering* PP (July 2018), pp. 1–1. DOI: 10.1109/TSE.2018.2859926 (cit. on p. 11).

[DAC99] M. Dwyer, G. Avrunin, J. Corbett. "Patterns in property specifications for finite-state verification". In: *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. 1999, pp. 411–420. DOI: 10.1145/302405.302672 (cit. on pp. 1, 10, 11).

[FHW+21] S. Frank, A. Hakamia, L. Wagner, D. Kesim, J. von Kistowski, A. van Hoorn. *Scenario-based Resilience Evaluation and Improvement of Microservice Architectures: An Experience Report*. 2021 (cit. on pp. 14, 18, 26, 32, 50, 62).

[Fla] Flask. *Flask Documentation*. https://flask.palletsprojects.com/en/2.1.x/. (Accessed on 07/28/2022) (cit. on p. 39).

[GD18] Y. Gan, C. Delimitrou. "The Architectural Implications of Cloud Microservices". In: *IEEE Computer Architecture Letters* 17.2 (2018), pp. 155–158. DOI: 10.1109/LCA.2018.2839189 (cit. on p. 1).

[Gre] Gremlin. https://www.gremlin.com/. (Accessed on 07/28/2022) (cit. on p. 31).

[Inf] InfluxDB. *Open Source Time Series Database*. https://www.influxdata.com/products/influxdb-overview/. (Accessed on 07/28/2022) (cit. on p. 31).

[Jin] Jinja. *Jinja Documentation (3.1.x)*. https://jinja.palletsprojects.com/en/3.1.x/. (Accessed on 07/28/2022) (cit. on p. 42).

[KHFH20] D. Kesim, A. van Hoorn, S. Frank, M. Häussler. "Identifying and Prioritizing Chaos Experiments by Using Established Risk Analysis Techniques". In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. 2020, pp. 229–240. DOI: 10.1109/ISSRE5003.2020.00030 (cit. on pp. 26, 32).

[Lit] Litmus. *Litmus Documentation*. https://docs.litmuschaos.io/. (Accessed on 07/28/2022) (cit. on p. 31).

[LS09] M. Leucker, C. Schallhart. "A brief account of runtime verification". In: *The Journal of Logic and Algebraic Programming* 78.5 (2009). The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07), pp. 293–303. ISSN: 1567-8326. DOI: https://doi.org/10.1016/j.jlap.2008.08.004. URL: https://www.sciencedirect.com/science/article/pii/S1567832608000775 (cit. on pp. 12, 13, 28).

[Mat] Matplotlib. https://matplotlib.org/. (Accessed on 07/28/2022) (cit. on p. 42).

[Mic] MicroK8s. *MicroK8s Documentation*. https://microk8s.io/docs. (Accessed on 07/28/2022) (cit. on p. 50).

[Mil19] R. Miles. *Learning chaos engineering : discovering and overcoming system weaknesses through experimentation / Russ Miles*. eng. First edition. Beijing: O'Reilly, 2019. ISBN: 1-4920-5099-7 (cit. on pp. 7, 8, 25).

[New15] S. Newman. *Building microservices : designing fine-grained systems*. Sebastopol, CA: O'Reilly Media, 2015. ISBN: 9781491950357 (cit. on pp. 1, 51).

[Pos] Postman. *Postman API Platform*. https://www.postman.com/. (Accessed on 07/28/2022) (cit. on p. 49).

[Pri] Principles of chaos engineering. https://principlesofchaos.org/. (Accessed on 07/28/2022) (cit. on p. 7).

[Pro] Prometheus. *Monitoring system & time series database*. https://prometheus.io/. (Accessed on 07/28/2022) (cit. on p. 31).

[Pyt] Python. *Homepage*. https://www.python.org/. (Accessed on 07/28/2022) (cit. on p. 31).

[RJ20] C. Rosenthal, N. Jones. *Chaos Engineering - System Resiliency in Practice*. Sebastopol: O'Reilly Media, 2020. ISBN: 978-1-492-04386-7 (cit. on pp. 1, 7, 25, 26, 32).

[Ulua] D. Ulus. *A pure Python package to monitor formal specifications over temporal sequences*. https://github.com/doganulus/python-monitors. (Accessed on 07/28/2022) (cit. on pp. 18, 39, 40).

[Ulub] D. Ulus. *Reelay Monitors*. https://doganulus.github.io/reelay/. (Accessed on 07/28/2022) (cit. on p. 40).

[Ulu19a] D. Ulus. "Timescales: A Benchmark Generator for MTL Monitoring Tools". In: Oct. 2019, pp. 402–412. ISBN: 978-3-030-32078-2. DOI: 10.1007/978-3-030-32079-9_25 (cit. on pp. 10, 41, 48).

[Ulu19b] D. Ulus. "Online Monitoring of Metric Temporal Logic using Sequential Networks". In: *CoRR* abs/1901.00175 (2019). arXiv: 1901.00175. URL: http://arxiv.org/abs/1901.00175 (cit. on pp. 10, 13, 18, 39–41).

[VFH14] V. Venikov, D. Fry, W. Higinbotham. *Transient Phenomena in Electrical Power Systems: International Series of Monographs on Electronics and Instrumentation, Vol. 24*. v. 24. Elsevier Science, 2014. ISBN: 9781483222714. URL: https://books.google.de/books?id=czWoBQAAQBAJ (cit. on pp. 1, 8).

[VGC+15] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, S. Gil. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud". In: *2015 10th Computing Colombian Conference (10CCC)*. IEEE. 2015, pp. 583–590 (cit. on p. 1).

[Wag21] L. Wagner. "Simulating Scenario-based Chaos Experiments for Microservice Architectures". Bachelor's Thesis. University of Stuttgart, 2021 (cit. on pp. 47, 56).

[Zah22] D. Zahariev. *Supporting and Verifying Transient Behavior Specifications in Chaos Engineering - Supplementary Materials*. July 2022. DOI: 10.5281/zenodo.6845089 (cit. on pp. 5, 49, 52, 54–57).

All links were last followed on July 28, 2022.

A Evaluation Resources

A.1 Chaos Experiment Evaluation - Chaos Experiments

Listing A.1 Evaluation - Chaos Experiment 1 (Without Transient Behavior Hypothesis Probe).

```
{
   "title":"Terminate one service1 pod",
   "description":"Terminating one service1 pod and observing steady-state hypothesis and transient behavior",
   "tags":["kubernetes"],
   "steady-state-hypothesis":{
      "title":"Verifying service remains healthy",
      "probes":[
         {
            "name":"deployment-is-fully-available",
            "type":"probe",
            "tolerance":true,
            "provider":{
               "type":"python",
               "module":"chaosk8s.probes",
               "func":"deployment_fully_available",
               "arguments":{
                  "name":"Service1"
               }
            }
         },
         ....
      ]
   },
   "method":[
      {
         "type":"action",
         "name":"terminate-service1-pod",
         "provider":{
            "type":"python",
            "module":"chaosk8s.pod.actions",
            "func":"terminate_pods",
            "arguments":{
               "name_pattern":"Service1",
               "rand":true
            }
         },
         "pauses":{
            "before":30,
            "after":90
         }
      }
   ]
}
```

Listing A.2 Evaluation - Chaos Experiment 1 (Transient Behavior Hypothesis Probe).

```
{
   "name":"check-transient-behaviors",
   "type":"probe",
   "tolerance":{
      "type":"jsonpath",
      "path":"$.result",
      "target":"body",
      "expect":[
         "True"
      ]
   },
   "provider":{
      "type":"http",
      "url":"http://localhost:5000/monitor",
      "method":"POST",
      "headers":{
         "Content-Type":"application/json"
      },
      "arguments":{
         "behavior_description":"Always when the instance count of a Service1 drops from 2 to
↪ 1, a new instance has to be spawned in max 30 sec",
         "specification":"always(((instances_dead(instances) and
↪ (resp_time_normal(resp_time)) ) since[0,30] instances_normal(instances)))",
         "specification_type":"mtl",
         "predicates_info":[{
               "predicate_name":"resp_time_normal",
               "predicate_logic":"smallerEqual",
               "predicate_comparison_value":"100"
            },{
               "predicate_name":"instances_normal",
               "predicate_logic":"equal",
               "predicate_comparison_value":"2"
            },{
               "predicate_name":"instances_dead",
               "predicate_logic":"smaller",
               "predicate_comparison_value":"2"
            }],
         "measurement_source":"influx",
         "measurement_points":[{
               "measurement_name":"resp_time",
               "measurement_query":"SELECT \"AvgResponseTime\" FROM
↪ \"TimeBatchRuns\".\"autogen\".\"Batch_Time\""
            },{
               "measurement_name":"instances",
               "measurement_query":"SELECT \"counts\" FROM
↪ \"AliveInstances\".\"autogen\".\"InstanceCounts\""
            }]
      }
   }
}
```

Listing A.3 Evaluation - Chaos Experiment 2 (Without Transient Behavior Hypothesis Probe).

```
{
   "title":"Increasing workload scenario",
   "description":"Increasing load and observing steady-state hypothesis and transient behavior",
   "tags":[
      "kubernetes"
   ],
   "steady-state-hypothesis":{
      "title":"Verifying service remains healthy",
      "probes":[
         {
            "name":"deployment-is-fully-available",
            "type":"probe",
            "tolerance":true,
            "provider":{
               "type":"python",
               "module":"chaosk8s.probes",
               "func":"deployment_fully_available",
               "arguments":{
                  "name":"Service1"
               }
            }
         },
         .....
      ]
   },
   "method":[
      {
         "type":"action",
         "name":"do-nothing",
         "provider":{
            "type":"python",
            "module":"chaosk8s.probes",
            "func":"deployment_fully_available",
            "arguments":{
               "name":"Service1"
            }
         },
         "pauses":{
            "after":600
         }
      }
   ]
}
```

Listing A.4 Evaluation - Chaos Experiment 2 (Transient Behavior Hypothesis Probe).

```
{
   "name":"check-transient-behaviors",
   "type":"probe",
   "tolerance":{
      "type":"jsonpath",
      "path":"$.result",
      "target":"body",
      "expect":["True"]
   },
   "provider":{
      "type":"http",
      "url":"http://localhost:5000/monitor",
      "method":"POST",
      "headers":{
         "Content-Type":"application/json"
      },
      "arguments":{
         "specification":"Globally, if {resp_time_high(resp_time)} [has occurred] then in response {instance_increase(instance_count)} [eventually holds] between 30 and 60 time units.",
         "specification_type":"psp",
         "predicates_info":[
            {
               "predicate_name":"resp_time_high",
               "predicate_description":"Response time is higher than 100ms",
               "predicate_comparison_value":"100",
               "predicate_logic":"biggerEqual"
            },
            {
               "predicate_name":"instance_increase",
               "predicate_description":"Instance counts increase",
               "predicate_comparison_value":"2",
               "predicate_logic":"bigger"
            }],
         "measurement_source":"influx",
         "measurement_points":[
            {
               "measurement_name":"resp_time",
               "measurement_query":"SELECT \"AvgResponseTime\" FROM \"TimeBatchRuns\".\"autogen\".\"Batch_Time\""
            },
            {
               "measurement_name":"instance_count",
               "measurement_query":"SELECT \"counts\" FROM \"AliveInstances\".\"autogen\".\"InstanceCounts\""
            }]
      }
   }
}
```

Listing A.5 Evaluation - Chaos Experiment 3 (Without Transient Behavior Hypothesis Probe).

```
{
   "title":"Increasing workload scenario",
   "description":"Increasing load and observing steady-state hypothesis and transient behavior",
   "tags":[
      "kubernetes"
   ],
   "steady-state-hypothesis":{
      "title":"Verifying service remains healthy",
      "probes":[
         {
            "name":"deployment-is-fully-available",
            "type":"probe",
            "tolerance":true,
            "provider":{
               "type":"python",
               "module":"chaosk8s.probes",
               "func":"deployment_fully_available",
               "arguments":{
                  "name":"Service1"
               }
            }
         },
         .....
      ]
   },
   "method":[
      {
         "type":"action",
         "name":"do-nothing",
         "provider":{
            "type":"python",
            "module":"chaosk8s.probes",
            "func":"deployment_fully_available",
            "arguments":{
               "name":"Service1"
            }
         },
         "pauses":{
            "after":600
         }
      }
   ]
}
```

Listing A.6 Evaluation - Chaos Experiment 3 (Transient Behavior Hypothesis Probe).

```
{
   "name":"check-transient-behaviors",
   "type":"probe",
   "tolerance":{
      "type":"jsonpath",
      "path":"$.result",
      "target":"body",
      "expect":["True"]
   },
   "provider":{
      "type":"http",
      "url":"http://localhost:5000/monitor",
      "method":"POST",
      "headers":{
         "Content-Type":"application/json"
      },
      "arguments":{
         "behavior_description":"description",
         "specification":"always(load_normal(service1_1_load) and
↪ load_normal(service1_2_load))",
         "specification_type":"mtl",
         "predicates_info":[
            {
               "predicate_name":"load_normal",
               "predicate_description":"description",
               "predicate_comparison_value":"0.8",
               "predicate_logic":"smaller"
            }
         ],
         "measurement_source":"prometheus",
         "measurement_points":[{
               "measurement_name":"service1_1_load",
               "measurement_query":" (rate (container_cpu_usage_seconds_total{
↪ pod='Service1-588bdc5d88-tkclz'}[1m])) ",
               "start_time":"10m",
               "end_time":"now",
               "steps":"1"
            },{
               "measurement_name":"service1_2_load",
               "measurement_query":" (rate
↪ (container_cpu_usage_seconds_total{pod='Service1-588bdc5d88-bvfn5'}[1m])) ",
               "start_time":"10m",
               "end_time":"now",
               "steps":"1"
            }
         ]
      }
   }
}
```

A.2 Compatibility Evaluation Resources

Listing A.7 MiSim Architecture Definition.

```
{
   "network_latency":".02+.002-0.001",
   "microservices":[
      {
         "name":"gateway",
         "instances":1,
         "capacity":10000,
         "loadbalancer_strategy":"even",
         "operations":[
            {
               "name":"API_Endpoint",
               "demand":1,
               "dependencies":[
                  {
                     "service":"service1",
                     "operation":"dependentCalculation"
                  }
               ]
            }
         ]
      },
      {
         "name":"service1",
         "instances":2,
         "patterns":[],
         "capacity":200,
         "operations":[
            {
               "name":"dependentCalculation",
               "demand":10,
               "dependencies":[
                  {
                     "operation":"service2.doWork",
                     "probability":1
                  }
               ]
            }
         ],
```

Listing A.8 MiSim Architecture Definition (Continuation).

```
        "s_patterns":[
            {
               "type":"autoscaling",
               "config":{
                  "period":0.5
               },
               "strategy":{
                  "type":"reactive",
                  "config":{
                     "lower_bound":0.2,
                     "upper_bound":0.8
                  }
               }
            }
         ]
      },
      {
         "name":"service2",
         "instances":1,
         "capacity":32000,
         "operations":[
            {
               "name":"doWork",
               "demand":1
            }
         ]
      }
   ]
}
```

Listing A.9 MiSim Experiment Definition.

```
{
  "name": "Minimal Scenario",
  "duration": 120,
  "seed": 42,
  "artifact": "gateway",
  "component": "ALL ENDPOINTS",
  "stimulus": "LOAD~ ./Examples/PaperExample/constant_loadArrivalRates_120sec.csv AND KILL service1 1@30"
}
```

Listing A.10 Compatibility Evaluation - Transient Behavior Specification 1.

```json
{
    "specification": " always(instance_dead(instance_count) since[0,3] instance_alive(instance_count)) ",
    "specification_type": "mtl",
    "predicates_info": [
        {
            "predicate_name": "instance_alive",
            "predicate_comparison_value": "2",
            "predicate_logic": "equal"
        },
        {
            "predicate_name": "instance_dead",
            "predicate_comparison_value": "1",
            "predicate_logic": "equal"
        }
    ],
    "measurement_source": "csv",
    "measurement_points": [
        {
            "measurement_name": "instance_count",
            "measurement_column": "instance_count"
        }
    ]
}
```

Listing A.11 Compatibility Evaluation - Transient Behavior Specification 2.

```
{
   "specification":"Before {instance_up(instance_count)}, it is never the case that {resp_time_high(resp_time)} [holds].",
   "specification_type":"psp",
   "predicates_info":[
      {
         "predicate_name":"instance_up",
         "predicate_description":"A new instance has been spawned.",
         "predicate_logic":"trendUpwardStrict"
      },
      {
         "predicate_name":"resp_time_high",
         "predicate_description":"Response time is too high when higher than 0.5.",
         "predicate_comparison_value":"0.5",
         "predicate_logic":"bigger"
      }
   ],
   "measurement_source":"csv",
   "measurement_points":[
      {
         "measurement_name":"instance_count",
         "measurement_column":"instance_count"
      },
      {
         "measurement_name":"resp_time",
         "measurement_column":"resp_time"
      }
   ]
}
```

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature