BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

VERIFICATION OF PARAMETRIC PROPERTIES OVER PROGRAM EXECUTION LOGS

OVĚŘOVÁNÍ PARAMETRICKÝCH VLASTNOSTÍ NAD ZÁZNAMY BĚHŮ PROGRAMŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

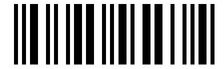
AUTHOR
AUTOR PRÁCE
Bc. JAKUB ŠURÁŇ

SUPERVISOR
VEDOUCÍ PRÁCE
Ing. ALEŠ SMRČKA, Ph.D.

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment



158136

Institut:	Department of Intelligent Systems (DITS)
Student:	**Šuráň Jakub, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Information Systems and Databases
Title:	**Verification of Parametric Properties over Program Execution Logs**
Category:	Software analysis and testing
Academic year:	2024/25

Assignment:

1. Familiarise yourself with runtime verification methods. Study the Plogchecker tool from the Testos platform. Examine the possibilities of specifying event sequences based on automata.
2. Identify the weaknesses of the current Plogchecker tool. Propose modifications to Plogchecker that enable efficient verification of compliance or violation of parametric event sequences. Focus on extending support for various parameter data types (e.g., strings, numbers, date and time).
3. Implement the newly designed tool. The key characteristics of the implementation should be reliability, efficiency, and maintainability.
4. Verify the functionality of the new tool by using automated tests for core functionalities. Evaluate the efficiency of the implementation through a set of performance tests considering the number of parameter values and the number and complexity of sequences.

Literature:

- Čaládi, F. (2022). Ověřování parametrických vlastností nad záznamy běhů programů. Brno. Vysoké učení technické v Brně. Fakulta informačních technologií.
- Havelund, K., Reger, G., Thoma, D., & Zalinescu, E. (2018). Monitoring Events that Carry Data. In *Lectures on Runtime Verification* (pp. 61-10)

Requirements for the semestral defence:
The first two points of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Smrčka Aleš, Ing., Ph.D.**
Head of Department:	Kočí Radek, Ing., Ph.D.
Beginning of work:	1.11.2024
Submission deadline:	31.7.2025
Approval date:	1.4.2025

Abstract

The primary objective of this thesis is the development of a tool for runtime verification of parametric properties—*Plogchecker 3.0*. This tool represents a reimplementation and significant enhancement of its predecessor, *Plogchecker 2.0*. Based on a thorough analysis of the earlier version's limitations, an improved monitoring algorithm and advanced resource management techniques have been designed. The new algorithm enables efficient retrieval of relevant monitor instances through optimized indexing structures. It also supports multiple monitoring modes, each allowing a different level of strictness. Beyond discarding resolved monitor instances, *Plogchecker 3.0* introduces preemptive discarding based on various victim selection strategies, improving scalability and memory utilization. Expressiveness has been extended through support for additional parameter data types and new property specification operators. The correctness of *Plogchecker 3.0* was verified through comprehensive functional and performance testing. These tests confirmed that the implementation meets all functional and non-functional requirements, and demonstrated strong scalability with respect to property complexity and quantity, as well as the parameter count.

Abstrakt

Hlavním cílem této diplomové práce je vytvoření nástroje pro verifikaci parametrických vlastností za běhu – *Plogchecker 3.0*. Tento nástroj představuje přepracovanou a vylepšenou verzi svého předchůdce, *Plogchecker 2.0*. Na základě podrobné analýzy jeho omezení byl navržen nový monitorovací algoritmus a pokročilé techniky pro správu systémových prostředků. Nový algoritmus je založen na efektivním vyhledávání relevantních instancí monitorů pomocí optimalizovaných indexovacích struktur. Dále podporuje několik režimů monitorování, které umožňují různou míru přísnosti při vyhodnocování. Kromě uvolňování již rozhodnutých instancí monitorů zavádí *Plogchecker 3.0* také preemptivní uvolňování na základě různých strategií výběru obětí. Tím je zajištěno lepší využití paměti a vyšší škálovatelnost. Vyjadřovací schopnosti nástroje byly rozšířeny o podporu nových datových typů pro parametry a nových operátorů pro specifikaci vlastností. Správnost implementace byla ověřena rozsáhlou sadou funkčních a výkonnostních testů. Ty prokázaly, že implementace splňuje všechny funkční i nefunkční požadavky. Zároveň potvrdily velmi dobrou škálovatelnost nástroje – jak vzhledem ke složitosti a počtu vlastností, tak i k počtu parametrů.

Keywords

runtime verification, parametric properties, Plogchecker, indexing tree, monitor instance, parameter instance, garbage collecting

Klíčová slova

verifikace za běhu, parametrické vlastnosti, Plogchecker, indexovací strom, instance monitoru, instance parametrů, garbage collecting

Reference

ŠURÁŇ, Jakub. *Verification of Parametric Properties over Program Execution Logs.* Brno, 2025. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, Ph.D.

Rozšířený abstrakt

Důkladné testování a prevence chyb v produktech představují kritickou oblast napříč všemi technologickými doménami. Odhalení defektů a nesrovnalostí již během vývoje často vede k výrazným finančním i materiálním úsporám a je proto vysoce žádoucí. To platí beze zbytku i v oblasti softwarového inženýrství, která využívá různé přístupy k tomuto účelu – například formální verifikaci či dynamickou analýzu. Oba zmíněné přístupy však mají svá omezení. U formální verifikace jde především o vysoké výpočetní nároky při aplikaci na rozsáhlejší systémy, zatímco dynamická analýza nedokáže nabídnout absolutní formální záruky správnosti.

Přijatelný kompromis mezi těmito dvěma metodami představuje verifikace za běhu (*runtime verification – RV*). Stejně jako formální verifikace umožňuje ověřovat formálně definované vlastnosti, avšak výhradně na základě jedné konkrétní exekuce programu. Díky tomu kombinuje formální přesnost s nízkými nároky dynamické analýzy. Nástroje *RV* jsou typicky založeny na tzv. monitorech – speciálních strukturách, které zpracovávají tok událostí ze zkoumaného systému a průběžně hlásí porušení specifikovaných vlastností. V posledních letech byl tento přístup dále rozšířen o podporu parametrických událostí – tedy událostí nesoucích konkrétní hodnoty parametrů.

Tato diplomová práce se zaměřuje na rozvoj a vylepšení nástroje pro verifikaci parametrických vlastností za běhu – *Plogchecker 2.0*. Úvodní část práce je věnována vymezení klíčových pojmů v oblasti parametrické *RV*. Následně jsou porovnány různé přístupy používané v této oblasti, přičemž hlavní pozornost je věnována technice rozdělování sledované stopy (tzv. *trace slicing*). Tento přístup je podpořen představením několika existujících nástrojů, které jej využívají.

Další část práce se věnuje podrobné analýze současné verze nástroje *Plogchecker 2.0*. Zaměřuje se na identifikaci jeho slabin a návrh možných zlepšení. Nejzásadnější problémy byly identifikovány v monitorovacím algoritmu, který je poměrně nestandardní, složitý a obtížně udržovatelný. Navíc v některých mezních případech vykazuje nesprávné chování.

Za účelem odstranění nalezených nedostatků byla navržena nová a vylepšená verze nástroje – *Plogchecker 3.0*. Jejím hlavním přínosem je nově navržený monitorovací algoritmus, který využívá efektivní indexovací stromové struktury pro rychlé vyhledání relevantních instancí monitorů dle konkrétních parametrických hodnot. Algoritmus navíc podporuje dva režimy vyhodnocování – standardní režim (*Standard*) a režim bez povolení událostí mimo pořadí (*No Out-of-Order Event – NOoOE*). Každý z nich se vyznačuje jinou mírou přísnosti. Výrazného zlepšení se dočkalo také hospodaření s pamětí – nástroj kromě běžného uvolňování již rozhodnutých instancí podporuje i preemptivní uvolňování na základě různých strategií výběru obětí. Vyjadřovací schopnosti byly dále rozšířeny o nové datové typy pro parametry a nové operátory pro specifikaci vlastností.

Nástroj *Plogchecker 3.0* byl následně implementován podle navržené architektury. Skládá se z několika paralelně běžících služeb – některé se starají o samotné monitorování (např. zpracování událostí a správa instancí), jiné o efektivní správu prostředků (např. prořezávání indexovacích stromů a preemptivní uvolňování).

Implementace nástroje byla důkladně ověřena pomocí rozsáhlých funkčních a výkonnostních testů. Funkční testy se zaměřily na ověření všech požadovaných funkčních i nefunkčních požadavků. Ověřena byla rovněž přenositelnost nástroje na různé cílové platformy.

Výkonnostní testování probíhalo ve dvou fázích. V první fázi bylo cílem otestovat škálovatelnost nástroje vzhledem ke komplexitě a počtu vlastností, ale i počtu parametrů. Oba monitorovací režimy vykazovaly téměř totožnou paměťovou náročnost (rozdíl do 2 %). U doby běhu byl však režim *NOoOE* pomalejší až o 10 %, což odpovídá jeho vyšší přísnosti.

Testy potvrdily, že nástroj dobře škáluje s rostoucí složitostí i počtem vlastností – běhová doba i paměťová náročnost rostly téměř lineárně. U testů s rostoucím počtem parametrů byl růst nejprve rovněž lineární (do cca 8 parametrů), poté však došlo k prudkému nárůstu způsobenému stavovou explozí počtu instancí.

Druhá fáze testování se zaměřila na srovnání různých strategií preemptivního uvolňování instancí. Nejlépe si vedla strategie *LRU*, která nabízela dobrý kompromis mezi úsporou paměti a minimalizací ztracených porušení. Dobře si vedla i strategie náhodného výběru (*Random*). Nejhorších výsledků dosáhla strategie *LFU*, která měla nejvyšší počet ztracených porušení.

Verification of Parametric Properties over Program Execution Logs

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Aleš Smrčka, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Jakub Šuráň
July 30, 2025

Acknowledgements

I would like to thank my supervisor Ing. Aleš Smrčka, Ph.D. for his help and guidance during the work on this thesis. I would also like to thank to my loved ones for their patience and support over the course of my studies.

Contents

1 Introduction **3**

2 Runtime Verification with Parametric Properties **5**
- 2.1 Approaches to the Parametric Runtime Monitoring . . . 5
- 2.2 Trace Slicing . . . 6
- 2.3 Formalism for the Properties Specification . . . 9
- 2.4 Monitor Generation . . . 10

3 Existing Tools for Parametric Runtime Verification **15**
- 3.1 JavaMOP . . . 15
- 3.2 Tracematches . . . 18
- 3.3 Larva . . . 20

4 Plogchecker 2.0 **23**
- 4.1 Overview . . . 23
- 4.2 Property Specification . . . 24
- 4.3 Monitoring Algorithm . . . 27
- 4.4 Garbage Collecting . . . 28

5 Enhancements Proposal **31**
- 5.1 Requirements . . . 31
- 5.2 New Design Overview . . . 31
- 5.3 Property Specification . . . 33
- 5.4 New Design of the Monitoring Algorithm . . . 36
- 5.5 Monitoring Modes . . . 42
- 5.6 Improvements of the Garbage Collecting . . . 45

6 Implementation Details **51**
- 6.1 Overview . . . 51
- 6.2 Configuration Parser . . . 52
- 6.3 Property Watchers . . . 56
- 6.4 Services . . . 59

7 Testing and Evaluation **64**
- 7.1 Scaling Benchmarks . . . 64
- 7.2 Comparison of Garbage Collection Strategies . . . 69
- 7.3 Functional Verification . . . 78

8 Conclusion **81**

Bibliography **83**

A Contents of the Attached Memory Media **85**

B Semantic Checks for Binary Operators in Constraints **86**

C Full Results of the Garbage Collection Strategies Comparison **88**

Chapter 1

Introduction

Testing and verifying the correct functionality of systems is a critical activity across all technological domains. Unexpected bugs and failures can lead to substantial financial losses and, in extreme cases, even fatalities. As a result, detecting and resolving unintended or faulty behavior is a vital part of the system development lifecycle—particularly in computer science. Various approaches exist to detect errors in hardware and software systems and to verify their correctness.

One approach involves the use of formal verification methods, such as model checking or theorem proving. These techniques can formally prove the correctness of a system with respect to a given formal specification. However, while they provide absolute mathematical guarantees, they are often computationally expensive. Additionally, defining an appropriate formal model for a system can be challenging and error-prone. Another approach is dynamic program analysis, which examines system behavior during individual executions using specific test cases. However, the scope of analysis in a single test case is limited, necessitating the use of a comprehensive test suite. To ensure adequate coverage, an appropriate coverage criterion must be applied.

Runtime verification (*RV*) offers a practical compromise between these two approaches. Like formal verification, *RV* checks whether system executions conform to specified formal properties. However, it avoids the high computational cost by employing dynamic analysis during a single execution. *RV* tools typically use monitors—runtime constructs that observe events emitted by the system, track relevant states and flag violations of expected behavior. In recent years, parametric runtime verification has gained traction. It extends *RV* by supporting data-carrying events, allowing a single property to cover many parameterized cases, thereby broadening *RV*'s applicability.

The primary objective of this thesis is to reimplement and optimize *Plogchecker 2.0*, a tool for *parametric runtime verification*. This tool is developed as part of the Testos[1] platform. The work aims to analyze the current version, identify its limitations and design an improved version—*Plogchecker 3.0*. The focus is on enhancing performance and scalability, with the goal of enabling use in real-world applications. This is achieved through a new monitoring algorithm featuring several optimizations and operational modes offering different levels of monitoring strictness. Expressiveness is also improved by supporting additional data types for event parameters.

The remainder of this thesis is organized as follows. Chapter 2 introduces the formal foundations of *parametric runtime verification*. This is followed by Section 3, which re-

[1] A modular platform for testing tools—http://testos.org/.

views several state-of-the-art tools based on the *trace slicing* approach. Chapter 4 presents the architecture and limitations of *Plogchecker 2.0*, while Chapter 5 outlines the proposed improvements that led to the development of *Plogchecker 3.0*. Chapter 6 describes the implementation details of the new tool. Chapter 7 then presents a thorough evaluation of the system, covering both functional and performance aspects, and includes a comparison of the supported monitoring modes. Finally, Chapter 8 concludes the thesis by summarizing the key contributions and suggesting potential directions for future work.

Chapter 2

Runtime Verification with Parametric Properties

This chapter lays the formal groundwork for the thesis within the field of *runtime verification*. It begins in Section 2.1 with an introduction to *runtime verification*, including the concept of *parametric runtime verification* and an overview of common implementation approaches. Section 2.2 focuses on *trace slicing*, the approach most relevant to this work. Section 4.2 reviews various formalisms for specifying properties, and Section 2.4 details the methods for generating monitors from these specifications.

2.1 Approaches to the Parametric Runtime Monitoring

Runtime verification (*RV*) [9] is a dynamic analysis technique, meaning the system is analyzed during a single execution. Its primary goal is to check whether the system satisfies a user-defined correctness property. These properties are specified using a chosen formal specification language (for a detailed discussion, see Section 4.2). The core component of *RV* is the monitor—a decision procedure for the specified property. The monitor processes traces, which are sequences of events produced by the system during execution. A typical *RV* process consists of three main phases:

1. generating monitors from the property specification,
2. instrumenting the analyzed system to produce relevant events,
3. analyzing the system execution using the monitor.

Originally, *RV* tools were designed to handle only properties over propositional events (i.e., events without associated data). However, in recent years, parametric events (i.e., events that carry data) have become the central focus of research in the field. According to Havelund et al. [13], there are five distinct approaches to parametric runtime verification:

Monitoring first-order temporal properties directly interprets properties specified using first-order temporal logic (FOTL). For example, the tool *MonPoly* [3] employs this approach. The core idea is to translate FOTL formulas into relational algebra. The input formula is first decomposed into subformulas, which are then converted into relational algebra expressions. These expressions are evaluated iteratively, ultimately producing the set of violations.

Monitoring modulo theories is a general framework based on a new logic called temporal data logic (TDL) [8]. TDL combines propositional temporal logic (e.g., linear temporal logic, or LTL) with first-order theories, resulting in a clear separation between the data and temporal aspects. The corresponding monitoring algorithm incrementally evaluates propositional temporal properties enhanced with satisfiability modulo theories (SMT) solving.

Parametric trace slicing monitors properties separately for each combination of parameter values in the trace. The algorithm splits the input trace into slices where each slice contains a distinct set of parameter bindings. Each property is then checked independently over each slice. This approach is discussed in detail in Section 2.2.

Rule-based monitoring draws on techniques widely used in artificial intelligence. Its main component is a rule system—a collection of rules, each consisting of a left-hand side (a list of conditions) and a right-hand side (an action). The monitor maintains a rule state (i.e., a set of facts) and fires a rule when all its conditions match facts in the current state. Rule execution typically modifies the fact database by adding or removing entries. A representative tool using this approach is *LogFire* [11].

Stream processing models system behavior using input streams and evaluates properties as Boolean output streams (verdicts). The *LOLA* framework [7] pioneered this approach by introducing a specification language for defining output streams based on their dependencies on the current, past or future values of other streams (including themselves).

Tools based on the *trace slicing* approach are currently considered the most efficient among existing runtime verification systems [13]. For this reason, the remainder of this thesis is primarily dedicated to this method.

2.2 Trace Slicing

This section defines the fundamental primitives used in the *trace slicing* approach. These include events and properties (initially non-parametric, then parametric) as well as the *trace slicing* concept itself. The discussion below primarily follows the terminology and notation from [4].

Definition 2.2.1 (Non-parametric events and traces)**.** Let E be a finite set of **events**. A **trace** is an element of E^*, i.e., any finite sequence of events from E. If an event $e \in E$ occurs in a trace $w \in E^*$, we write $e \in w$.

Example 2.2.1. Consider a simple resource representing a network client. The client can connect to a server, send messages, receive messages and disconnect. In this case,
$E = \{connect, send, receive, disconnect\}$, and execution traces for this resource are sequences such as `"connect disconnect"` or `"connect send receive send receive"`.

Definition 2.2.2 (Non-parametric property)**.** Let E^* be the set of traces as defined in Definition 2.2.1. A non-parametric **property** P is a function $P : E^* \to C$ that partitions the set of traces into categories C. In general, C can be any set, but it commonly includes categories such as *validating*, *violating*, and *don't know*.

Example 2.2.2. Consider the resource from the previous example. The specification S of its desired behavior can be provided, for instance, by the following regular expression—`connect (send receive)* disconnect`. This expression specifies that the client must first connect to the server, may then repeatedly send and receive messages in pairs, and must eventually disconnect.

Let the set of categories be $C = \{validating, violating, don't_know\}$. The traces matching the regular expression are classified *validating*, such as
`"connect disconnect"` or `"connect send receive disconnect"`.

Traces that do not match the specification fall into two subcategories:

- **Violating traces** which can never become valid regardless of future events. An example is `"connect send disconnect"`.
- **Uncertain traces** which are not yet valid but could become valid if additional events are observed. An example is `"connect send"`.

Given the specification S, the corresponding property P_S for the trace $w \in E^*$ is formally defined as:

$$P_S(w) = \begin{cases} validating & \text{if } w \in L(S) \\ violating & \text{if } (w \notin L(S)) \land \forall w' \in E^* : ww' \notin L(S) \\ don't\ know & \text{otherwise} \end{cases}$$

Definition 2.2.3 (Parameter instances and their least upper bound)**.** Let X be a set of parameters and let V be a set of corresponding parameter values. A partial function $\theta \in [X \rightharpoonup V]$ is called a **parameter instance**. Two instances θ and θ' are compatible iff:

$$\forall x \in Dom(\theta) \cap Dom(\theta') : \theta(x) = \theta'(x)$$

Compatible instances θ and θ' can be combined into their **least upper bound**, denoted $\theta \sqcup \theta'$, as follows:

$$(\theta \sqcup \theta')(x) = \begin{cases} \theta(x) & \text{if } \theta(x) \text{ } \textit{is defined} \\ \theta'(x) & \text{if } \theta'(x) \text{ } \textit{is defined} \\ undefined & \text{otherwise} \end{cases}$$

Example 2.2.3. The previous examples will be now extended to the parametric case. Assume that the studied resource is parameterized by both a client and a server. Let the set of parameters be defined as $X = \{c, s\}$, and let the corresponding set of parameter values be $V = \{a_0, \ldots, a_i, b_0, \ldots, b_j\}$. Sample parameter instances include for example, $\theta_1 = \langle c \mapsto c_0 \rangle$, $\theta_2 = \langle c \mapsto c_1 \rangle$ and $\theta_3 = \langle c \mapsto c_0, s \mapsto s_1 \rangle$. The instances θ_1 and θ_3 are compatible, whereas θ_1 and θ_2 are not. The least upper bound of θ_1 and θ_3 is $\theta_1 \sqcup \theta_3 = \langle c \mapsto c_0, s \mapsto s_1 \rangle$.

Definition 2.2.4 (Parametric events and traces)**.** Let X be a set of parameters together with their set of parameter values from Definition 2.2.3. If E is a set of events from Definition 2.2.1, then let $E(X)$ be a set of corresponding **parametric events** $e(\theta)$, where $e \in E$ and θ is a parameter instance from Definition 2.2.3. A **parametric trace** τ is a trace with events in $E(X)$, that is $\tau \in E(X)^*$.

Example 2.2.4. Consider that all events defined in Example 2.2.1 can be parametrized using the parameters from Example 2.2.3. To simplify notation, only parameter values will be listed in parameter instances from now on—i.e., instead of $\langle c \mapsto c_0, s \mapsto s_1 \rangle$, we will write $\langle c_0, s_1 \rangle$. A sample parametric trace might look as follows—"$\texttt{connect}(c_0, s_0)$ $\texttt{send}(c_0, s_0)$ $\texttt{connect}(c_1, s_0)$ $\texttt{receive}(c_0, s_0)$ $\texttt{disconnect}(c_1, s_0)$ $\texttt{disconnect}(c_0, s_0)$".

Definition 2.2.5 (Less informative relation for parameter instance)**.** Let θ and $\theta^{'}$ be the parameter instances as defined in Definition 2.2.3. We say that $\theta^{'}$ is **less informative** than θ, written as $\theta^{'} \sqsubseteq \theta$, iff:

$$\forall x \in X : \theta^{'}(x) \text{ is defined } \implies (\theta(x) \text{ is defined } \wedge \theta(x) = \theta^{'}(x))$$

Example 2.2.5. Assume parameter instances θ_1, θ_2 and θ_3 from Example 2.2.3. Both θ_1 and θ_2 are less informative than θ_3, that is $\theta_1 \sqsubseteq \theta_3$ and $\theta_2 \sqsubseteq \theta_3$.

Definition 2.2.6 (Trace slice)**.** Let $\tau, \tau' \in E(X)^*$ be parametric traces (Definition 2.2.4), θ, θ' parameter instances (Definition 2.2.3) and $e(\theta')$ a parametric event (Definition 2.2.4). Additionally, let $\cdot$ denote concatenation. Then, the **θ-trace slice** (or simply **trace slice**) $\tau{\restriction_\theta} \in E^*$ is the non-parametric trace recursively defined as:

1. **Base case**: $\varepsilon{\restriction_\theta} = \varepsilon$
2. **Recursive step**: $(\tau \cdot e(\theta')){\restriction_\theta} = \begin{cases} (\tau{\restriction_\theta}) \cdot e & \text{if } \theta^{'} \sqsubseteq \theta \\ \tau{\restriction_\theta} & \text{otherwise} \end{cases}$

where ε is the empty trace and $\sqsubseteq$ is the less informative relation for parameter instances from Definition 2.2.5.

Example 2.2.6. Intuitively, *trace slicing* $\tau{\restriction_\theta}$ first filters out all parametric events that are irrelevant to the parameter instance θ (i.e., those that are not compatible or are more informative). Then, the parameter annotations are discarded from the remaining events, effectively converting the original parametric trace into a non-parametric trace consisting only of relevant events.

For example, given the parametric trace "$\texttt{connect}(c_0, s_0)$ $\texttt{send}(c_0, s_0)$ $\texttt{connect}(c_1, s_0)$ $\texttt{receive}(c_0, s_0)$", the *trace slice* corresponding to the parameter instance $\langle c_0, s_0 \rangle$ is defined as "`connect send receive`".

Definition 2.2.7 (Parametric properties)**.** Let X be a set of parameters with the corresponding parameter values V as defined in Definition 2.2.3. Additionally, consider the non-parametric property $P : E^* \to C$ from Definition 2.2.2. Then, the parametric property $\Lambda X.P$ is defined as a function $E(X)^* \to [[X \rightharpoonup V] \to C]$ given by:

$$(\Lambda X.P)(\tau)(\theta) = P(\tau{\restriction_\theta})$$

where $\tau{\restriction_\theta}$ is the *trace slice* from Definition 2.2.6 and $\Lambda X.P$ represents a parametric lifting of P, i.e. high-order function that takes a parametric trace τ and returns a function from parameter instances to the verdict categories C.

Conceptually, the parametric property can be viewed as if multiple instances of the non-parametric property P are evaluated simultaneously over the parametric trace. Each instance corresponds to a specific parameter instance and focuses solely on the relevant events.

2.3 Formalism for the Properties Specification

This section briefly discusses the formalisms used to specify the properties to be monitored. Currently, no single formalism is widely regarded as superior within the runtime monitoring community [9]. As a result, the choice of formalism for a particular tool typically depends on the intended use case and the developers' subjective preferences.

Most tools employ formalisms that generate regular languages which are sufficient for the majority of use cases. Common strategies include the following:

Regular expressions (or extended regular expressions)
This method is used, for example, in *Tracematches* [1].

Finite-state automata
This approach is employed by the developers of *Larva* [6].

Temporal logic (especially variants of LTL)
The languages described by LTL belong to the class of star-free languages, making them less expressive than regular languages. Nevertheless, temporal logic often provides a more concise way to specify properties, making it appealing for many tools. For instance, the *DejaVu* tool uses this approach [12].

An interesting approach has been taken by the developers of the *JavaMOP* tool [19]. This tool employs a formalism-independent monitoring algorithm. As a result, it offers a variety of plugins that support the use of different formalisms for specifying monitoring properties. In fact, all of the approaches mentioned so far (i.e., regular expressions, finite-state automata and LTL) are supported by *JavaMOP*.

On the other hand, some tools employ formalisms that offer even greater expressiveness than regular languages. However, these advantages come at the cost of increased monitoring overhead, which may render such approaches unsuitable for many use cases. Commonly used techniques include:

Context-free grammar
JavaMOP supports this through a dedicated plugin.

Rule system
This strategy typically involves rewriting a set of facts using conditional rules. For example, it is used, in *LogFire* [11].

The original authors of *Plogchecker* chose to use regular expressions for specifying properties [21, 24]. Therefore, this formalism is the most relevant for the scope of this thesis. The remainder of this section is dedicated to defining the components of this specification approach.

Following [22], Definition 2.3.1 provides the formal specification of regular expressions.

Definition 2.3.1 (Regular expressions)**.** Let Σ be a finite alphabet. Moreover, consider the following language operations: $\cup$ (union), $\circ$ (concatenation) and * (Kleene star). Then, **regular expressions** (**RE**) over Σ are recursively defined as follows:

1. $\emptyset$ is a RE representing the empty language.

2. ε is a RE representing the language $\{\varepsilon\}$.

3. For any $a \in \Sigma$, a is a RE representing the language $\{a\}$.

4. Let R_1 and R_2 be regular expressions denoting the languages $L(R_1)$ and $L(R_2)$, respectively. Then the following operations also form REs (listed in order of increasing precedence):

   - *Alternation*: $(R_1 + R_2)$ is a RE representing the language $L(R_1) \cup L(R_2)$.
   - *Concatenation*: $(R_1 \cdot R_2)$ is a RE representing the language $L(R_1) \circ L(R_2)$.
   - *Iteration*: (R_1^*) is a RE representing the language $L(R_1)^*$.

In practice, the REs from the above formal definition are often slightly extended to allow easier usage. For example, the POSIX standard [23] defines several additional operators that simplify writing REs. Consider R to be a regular expression. The following additional operators are included:

- $^+$ (R^+ is equivalent to $R \cdot R^*$),
- ? ($R?$ is equivalent to $R + \varepsilon$),
- $\{m, n\}$ ($R\{m, n\}$ matches R at least m, but not more than n times),
- $\{m\}$ ($R\{m\}$ matches R exactly m times),
- $\{m,\}$ ($R\{m,\}$ matches R at least m times).

2.4 Monitor Generation

For runtime verification, it is essential to properly generate monitors which corresponds to the properties to be monitored. As explained in Section 4.2, *Plogchecker* uses regular expressions as the formalism for property specification. Therefore, finite automata are naturally used as the underlying formalism for monitor representation. This section covers the process of converting regular expression properties to the corresponding finite automaton monitors. The process consists of three steps:

1. Convert RE to NFA,
2. convert NFA to DFA,
3. minimize the DFA.

All these steps are well known and have been extensively studied in numerous publications. For the sake of completeness, they are further elaborated in dedicated subsections, by following [15].

Converting RE to NFA

The first step in generating a monitor from a regular expression property is the conversion to the corresponding non-deterministic finite automaton (NFA). The specification of the NFA is given in Definition 2.4.1.

Definition 2.4.1 (ε-NFA)**.** Let $\mathcal{P}(X)$ denote the power set of a set X. Then, a **non-deterministic finite automaton with ε-transitions** (or simply **ε-NFA**) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of states,
- Σ is a finite input alphabet,
- δ is a transition function $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states.

The conversion process is defined in Algorithm 2.4.1. First, the basic components of the regular expression are identified. Then, a corresponding NFA is created for each of them. Finally, these NFAs are progressively combined into a single automaton by applying the operators from the original regular expression.

Converting NFA to DFA

The next step in the monitor generation process is to derive the corresponding deterministic finite automaton (DFA) by converting the given non-deterministic finite automaton (NFA).

The formal specification of a deterministic finite automaton is provided in Definition 2.4.2.

Definition 2.4.2 (DFA)**.** A **deterministic finite automaton** (or simply **DFA**) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of states,
- Σ is a finite input alphabet,
- δ is a transition function $\delta : Q \times \Sigma \to Q$,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states.

The transition function δ of the DFA (Definition 2.4.2) is commonly extended to a variant $\hat{\delta}$ which accepts input strings $w \in \Sigma^*$ instead of single symbols $a \in \Sigma$—for a given state $q \in Q$:

1. **Base case**: $\hat{\delta}(q, \varepsilon) = q$
2. **Recursive step**: Let $w' \in \Sigma^*$, $a \in \Sigma$ and $w = w'a$. Then, $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, w'), a)$

Assume the ε-NFA from Definition 2.4.1. The ε-closure of a state $q \in Q$ is given in Definition 2.4.3.

Definition 2.4.3 (ε-closure)**.** Let $N = (Q, \Sigma, \delta, q_0, F)$ be a ε-NFA from Definition 2.4.1. The ε-closure of a state $q \in Q$ is defined as follows:

$$\varepsilon - closure(q) = \{q\} \cup \bigcup_{p \in \delta(q, \varepsilon)} \varepsilon - closure(p)$$

Algorithm 2.4.1 Conversion of RE to ε-NFA.

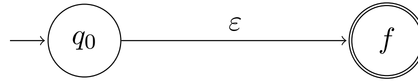
Input: Regular expression R over the alphabet Σ (as defined in Definition 2.3.1).
Output: An ε-NFA $N = (Q, \Sigma, \delta, q_0, F)$ (as defined in Definition 2.4.1).

1: Decompose R into its basic components: $\emptyset$, ε, and $a \in \Sigma$.

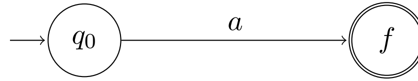
2: Construct an automaton for each of the basic components from the previous step using the following rules:

a. For the regular expression $\emptyset$, create an automaton that accepts the empty language:

b. For the regular expression ε, create an automaton that accepts the language $\{\varepsilon\}$:

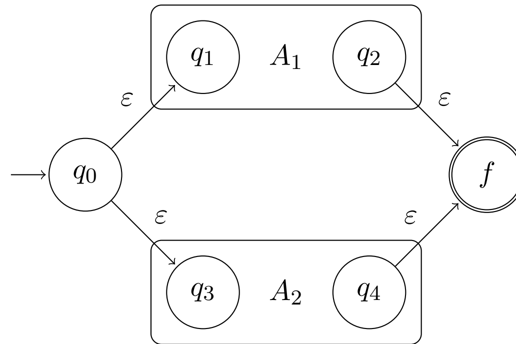


c. For the regular expression $a \in \Sigma$, create an automaton that accepts the language $\{a\}$:

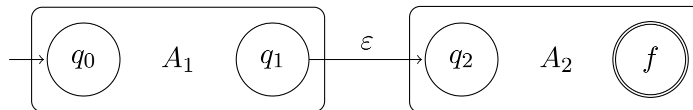


3: Recursively combine the automata for the basic components by applying the RE operations until a single automaton covering the entire original regular expression R is obtained. Let A_1 be an automaton accepting the language of R_1, and A_2 an automaton accepting the language of R_2. Use the following rules to combine them:

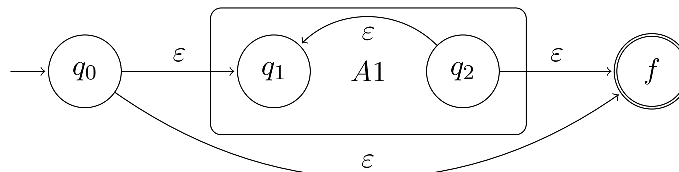
a. For the regular expression $R_1 + R_2$, create the following automaton:



b. For the regular expression $R_1 \cdot R_2$, create the following automaton:



c. For the regular expression R_1^*, create the following automaton:



The ε-closure (Definition 2.4.3) can be generalized to apply to a set of states rather than to a single state. The generalized form for $S \subseteq Q$ is defined as:

$$\varepsilon - closure(S) = \bigcup_{s \in S} \varepsilon - closure(s)$$

The procedure for converting an NFA to a DFA is described in Algorithm 2.4.2. It begins by constructing the initial state of the DFA as the ε-closure of the NFA's initial state. Then, it progressively defines the set of DFA states, where each DFA state corresponds to a set of NFA states. The algorithm adds only those states that are reachable from the DFA's initial state. The DFA's transition function is defined gradually. Finally, all DFA states that contain at least one final state of the original NFA are designated as final.

Algorithm 2.4.2 Conversion of ε-NFA to DFA.

Input: An ε-NFA $N = (Q, \Sigma, \delta, q_0, F)$.
Output: An DFA $D = (Q', \Sigma, \delta', q_0', F')$.

```
1:  q'_0 = ε − closure(q_0)                          ▷ Calculate new initial state
2:  Stack = {q'_0}, Q' = {q'_0}, δ' = {}
3:  while not empty(Stack) do
4:      for all a ∈ Σ do
5:          S ← pop(Stack)                           ▷ Assume S = {s_1, s_2, ..., s_k}
6:          R ← ⋃_{i=1}^{k} δ(s_i, a)
7:          S' ← ε − closure(R)
8:          δ'(S, a) ← S'          ▷ Progressively update the new transition function
9:          if S' ∉ Q' then
10:             Q' ← Q' ∪ {S'}         ▷ Progressively build the set of reachable states
11:             push(Stack, S')
12:         end if
13:     end for
14: end while
15: F' = {S ∈ Q' | S ∩ F ≠ ∅}                       ▷ Construct set of final states
16: return (Q', Σ, q'_0, δ', F')
```

Minimization of DFA

The final step in generating monitors is to minimize the resulting DFA. Various algorithms exist for this task. One of the oldest algorithms is Moore's minimization algorithm [20]. However, the performance of this algorithm depends more on the language accepted by the DFA than on the topology of the DFA itself. This limitation was later addressed by Hopcroft, who proposed a more efficient minimization algorithm [14]. The rest of the section is dedicated to the Hopcroft algorithm.

Before presenting the minimization procedure, it is necessary to define the equivalence relation between DFA states, as provided in Definition 2.4.4.

Definition 2.4.4 (State equivalence)**.** Consider the DFA defined in Definition 2.4.2. Two states $p, q \in Q$ are said to be **equivalent** (denoted $p \equiv q$) iff:

$$\forall w \in \Sigma^* : \hat{\delta}(p, w) \in F \iff \hat{\delta}(q, w) \in F$$

Steps for the DFA minimization according to Hopcroft are specified in Algorithm 2.4.3. It is based on the idea of partitioning the state-space Q by the state equivalence relation (Definition 2.4.4). The algorithm starts with the partitioning the states into two groups—final and non-final states. After that, it refines this partitioning iteratively based on how states transition under input symbols. At each step, it splits partitions to ensure that all states in a class are indistinguishable in terms of transitions to the other partitions. Finally, the algorithm constructs the minimal automaton by utilizing obtained set of equivalence classes of the original states $\mathcal{P}$.

Algorithm 2.4.3 Hopcroft's DFA minimization algorithm.

Input: A DFA $D = (Q, \Sigma, \delta, q_0, F)$.
Output: A minimized DFA $D' = (Q', \Sigma, \delta', q'_0, F')$.

1: $\mathcal{W} \leftarrow \{F, Q \setminus F\}$
2: $\mathcal{P} \leftarrow \{F, Q \setminus F\}$
3: **while** $\mathcal{W} \neq \emptyset$ **do**
4: $\quad S \leftarrow pick_one(\mathcal{W})$
5: $\quad \mathcal{W} \leftarrow \mathcal{W} \setminus \{S\}$
6: $\quad$ **for each** $a \in \Sigma$ **do**
7: $\quad\quad X \leftarrow \{q \in Q \mid \delta(q, a) \in S\}$
8: $\quad\quad$ **for each** p **in** $\mathcal{P}$ **do**
9: $\quad\quad\quad p_1 = p \cap X$
10: $\quad\quad\quad p_2 = p \setminus X$
11: $\quad\quad\quad$ **if** $p_1 \neq \emptyset \wedge p_2 \neq \emptyset$ **then**
12: $\quad\quad\quad\quad$ replace p in $\mathcal{P}$ with p_1 and p_2
13: $\quad\quad\quad\quad$ **if** $p \in \mathcal{W}$ **then**
14: $\quad\quad\quad\quad\quad$ replace p in $\mathcal{W}$ with p_1 and p_2
15: $\quad\quad\quad\quad$ **else**
16: $\quad\quad\quad\quad\quad$ add the smaller of p_1 and p_2 to $\mathcal{W}$
17: $\quad\quad\quad\quad$ **end if**
18: $\quad\quad\quad$ **end if**
19: $\quad\quad$ **end for**
20: $\quad$ **end for**
21: **end while**
22: Build a minimized DFA based on the obtained equivalence classes $\mathcal{P} = \{[q] \mid q \in Q\}$:

a. $Q' \leftarrow \mathcal{P}$

b. $\delta'([q], a) \leftarrow [\delta(q, a)]$ for each $[q] \in Q'$ and $a \in \Sigma$

c. $q'_0 \leftarrow [q_0] \in \mathcal{P}$

d. $F' \leftarrow \{[q] \in \mathcal{P} \mid [q] \cap F \neq \emptyset\}$

23: **return** $(Q', \Sigma, \delta', q'_0, F')$

Chapter 3

Existing Tools for Parametric Runtime Verification

This chapter provides an overview of three selected *parametric runtime verification* tools that employ the *trace slicing* approach. For each tool, the property specification method is detailed. Additionally, a concise explanation of its monitoring algorithm is provided. One of the most widely used tools, JavaMOP, is discussed in Section 3.1. Section 3.2 explores Tracematches, a tool that shares many similarities with JavaMOP but employs a distinct concept for its monitoring algorithm. Lastly, Section 3.3 focuses on Larva, which, unlike the other two tools, also supports the monitoring of real-time properties.

3.1 JavaMOP

JavaMOP [19] is a pioneering tool in the *trace slicing* approach. Unlike many similar tools, it employs a generic monitoring algorithm. The term generic here signifies that the tool is not limited to a single property specification formalism. Instead, JavaMOP supports multiple formalisms through plugins, such as finite-state automata, extended regular expressions (ERE), various variants of past-time linear temporal logic and others. Each property is translated into a corresponding internal representation, typically a finite-state automaton or a pushdown automaton, depending on the expressive power of the chosen formalism.

For example, consider the *Unsafe Iterator* property, a well-known case in the *runtime verification* community. This property describes invalid iterator usage in Java—*once an iterator for a collection is created, it cannot be used again if the underlying collection is modified.* The specification of this property using the JavaMOP ERE plugin is shown in Listing 3.1. It comprises three key components—the definition of events (including their parameters), the property pattern expressed as an ERE and the code to execute when the pattern matches. The `createI` event is the sole creation event, responsible for spawning new monitor instances.

```
Collection_UnsafeIterator(Collection c, Iterator i) {
  creation event createI after(collection c) returning(Iterator i) :
    call(Iterator Iterable+.iterator()) && target(c) {}
  event next before(iterator i):
    (
        call(* Iterator.hasNext(..)) ||
        call(* Iterator.next(..))
```

```
    ) && target(i) {}
  event updateC before(Collecton c) :
    (
        call(* Collection+add*(..)) ||
        call(Collection+remove*(..))
    ) && target(c) {}

  ere : createI next* updateC+ next

  @match { System.err.println("Unsafe usage of~iterator detected!"); }
}
```

Listing 3.1: Specification of the *unsafe iterator* property using JavaMOP ERE plugin (partially taken from [16]).

The monitoring algorithm relies on the concept of a parametric monitor which is created for each property to be monitored. A parametric monitor is a convenient data structure consisting of two main parts. The first is the base monitor which corresponds to the property definition. The second is a set of monitor instances, each responsible for tracking the *trace slice* associated with a specific parameter instance. Thus, the parametric monitor can be thought of as a collection of base monitor duplications running in parallel, with each duplication managing a unique *trace slice*.

Based on the discussion above, it is evident that the monitoring algorithm must handle numerous monitor instances for distinct parameter instances. To achieve this efficiently, it employs specialized indexing trees to retrieve instances associated with specific parameter instances. These trees are internally implemented as multi-level maps where each level represents the binding of one parameter. Parameter values serve as the keys, while monitor instances are the values.

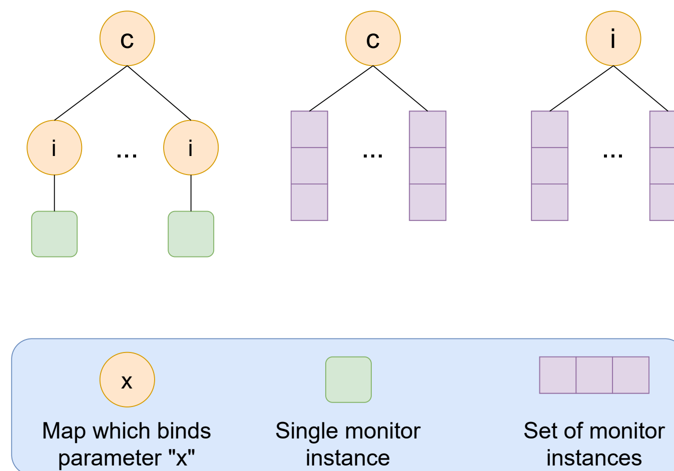


Figure 3.1: Example of indexing trees for the *unsafe iterator* property (partially taken from [16]).

JavaMOP uses several variants of these trees per property to account for all distinct subsets of parameters bound in its events. As a result, the value in the map can either be a concrete monitor instance when the parameter instance is fully bound, or a set of monitors corresponding to a partially bound parameter instance. An example of indexing trees for the *Unsafe Iterator* property is shown in Figure 3.1. It illustrates all three indexing trees—the leftmost tree is a 2-level map containing concrete monitor instances, while the remaining two trees are 1-level maps that hold sets of monitors as their values.

Algorithm 3.1.1 presents the simplified pseudocode of the monitoring algorithm. This procedure is executed for each received parametric event and consists of two main parts. In the first part, the algorithm determines whether a monitor instance for the given parameter instance already exists. If it does, no further action is required. However, if it does not exist, the algorithm attempts to create a new instance by cloning the most similar existing monitor instance. If no such instance is found and the received event is a creation event, a brand-new instance is created. Additionally, the algorithm may attempt to generate a new instance by merging the parameter instance from the event with parameter instances from existing monitors. In the second part, all monitor instances relevant to the parameter instance are updated accordingly.

Algorithm 3.1.1 Pseudocode of the monitoring algorithm used in JavaMOP (partially taken from [19]).

Global: A mapping of the parameter instances to monitor instances *pool*.
Initialization: $pool.clear()$ ▷ Mapping is cleared at the beginning
1: **function** PROCESS_EVENT($e(\theta)$)
2: **if not** $is_matching_monitor_instance_found(\theta)$ **then**
3: $most_similar = pool.find_most_similar_monitor_instance(\theta)$
4: **if** $most_similar$ **is** $defined$ **then**
5: $pool.add(most_similar.copy(\theta))$
6: **else if** $is_creation_event(e(\theta))$ **then**
7: $pool.add(create_new(\theta))$
8: **end if**
9: **end if**
10: **for each** θ' **in** $get_less_informative_parameter_instances(\theta)$ **do**
11: **for each** $monitor_instance$ **in** $pool.get_relevant(\theta')$ **do**
12: $\theta_m = monitor_instance.get_associated_parameter_instance()$
13: **if not** $pool.is_matching_monitor_instance_found(\theta_m \sqcup \theta)$ **then**
14: $pool.add(monitor_instance.copy(\theta_m \sqcup \theta))$
15: **end if**
16: **end for**
17: **end for**
18: **for each** $monitor_instance$ **in** $pool.get_relevant_monitor_instances(\theta)$ **do**
19: $monitor_instance.update(e(\theta))$
20: **end for**
21: **end function**

In addition to indexing trees, JavaMOP incorporates other optimizations to enhance efficiency [18]. For instance, it minimizes the number of monitor instances by generating only those that are strictly necessary. This decision is based on the events previously observed for the corresponding *trace slice*. Furthermore, JavaMOP employs caches to store

recently accessed monitor instances, with each indexing tree having its own dedicated cache. Another optimization involves merging indexing trees that share a common parameter prefix. For example, trees indexed by `(c, i)` and `(c)` can be merged, as they both include the `c` parameter at their first level. However, trees indexed by `(c, i)` and `(i)` cannot be merged because the parameters in their first level are different.

One of the most significant challenges during the monitoring process is managing memory usage effectively. To address this, JavaMOP employs a form of garbage collection for monitor instances [17]. First, it removes instances whose verdict has already been determined—whether they satisfy the property or definitively violate it. Additionally, it eliminates monitor instances that have no prospect of reaching a verdict. The parameter values in JavaMOP are, in fact, objects from the monitored program. When these objects are garbage collected by the program, their associated monitor instances may no longer be needed. To identify such cases, JavaMOP uses heuristics based on precomputed helper sets for each property. These heuristics help determine whether a specific instance is truly unnecessary and can be safely discarded.

3.2 Tracematches

Although Tracematches [1] is also based on *trace slicing*, it adopts a slightly orthogonal monitoring approach compared to JavaMOP. Rather than creating a monitor instance for each data binding, Tracematches uses a single monitor (in the form of a finite automaton) per property. The states of this monitor are annotated with constraints to properly represent all relevant data bindings. A constraint is a Boolean combination which binds parameters to concrete values. In Tracematches terminology, these bindings are referred to as *disjuncts*.

Tracematches use extended regular expressions (ERE) as the specification formalism for properties. More precisely, each property must be defined using a specific construct called a *tracematch*. A tracematch consists of three main components—a declaration of symbols, an ERE over these symbols (specifying the property) and a piece of code to be executed when the tracematch is satisfied. A match occurs when a trace (restricted to the events corresponding to the defined symbols) belongs to the language of the specified ERE. As an example, consider again the safe iterator property (see Section 3.1 for details). Its specification is shown in Listing 3.2.

```
tracematch (Iterator i, Collection c) {
    sym createI after returning(i):
        call(Iterator Collection.iterator()) && target(c);
    sym next before:
        call(Object Iterator.next()) && target(i);
    sym updateC after:
        call(* Collection.update(...)) && target(c);

    createI next* updateC+ next

    { throw new ConcurrentModificationException(); }
}
```

Listing 3.2: Specification of the *unsafe iterator* property in Tracematches (partially taken from [1]).

Before describing the monitoring algorithm, it is important to highlight a significant semantic difference between JavaMOP and Tracematches. In Tracematches, a single event received from the monitored program can correspond to multiple symbols. These symbols serve as logical building blocks used in the definition of the ERE. A symbol can thus be viewed as a function that maps events to constraints—effectively binding parameters to concrete values. If a symbol and an event are incompatible (e.g., if their parameter sets are disjoint), the symbol evaluates to *false*. If the received event does not match any symbol defined in the property, it is simply ignored.

The monitoring process is interpreted over a nondeterministic finite automaton (NFA) with a single final state, equivalent to the ERE defined in the specification. To simulate skipped events, a special symbol called *skip* is introduced. The automaton includes self-loops labeled with the *skip* symbol on all states except the final one. Additionally, each state is labeled with constraints representing the parameter bindings accumulated while processing the input trace. These labels are progressively updated as new events are received and mapped to constraints via the defined symbols. An automaton with appropriately labeled states corresponding to the property from Listing 3.2 is shown in Figure 3.2.

Figure 3.2: Example of the automaton for the *safe iterator* property. The current labels are placed under each state (partially taken from [2]).

The simplified pseudocode of the monitoring algorithm is shown in Algorithm 3.2.1. Initially, only the label on the initial state is set to *true*, while labels on all remaining states are set to *false*. The algorithm then processes the trace event by event. For each event, all corresponding symbols are iterated over. This results in eventual updates to the labels of the states whose transitions involve these symbols.

Since a single event can correspond to multiple symbols, the label update is performed in two steps. First, intermediate results in the form of disjuncts are accumulated for each state. Then, in the second step, the actual labels of the states are updated. If any new solutions are produced in the final state as a result of the updates, the corresponding action code from the specification is executed.

As an optimization, Tracematches utilizes indexing trees similar to those used in JavaMOP. These trees use parameter bindings as keys and store disjuncts as values [2]. This significantly speeds up event processing, as the monitoring algorithm can directly retrieve only those disjuncts relevant to the current event. The indexing tree is implemented as a multilevel map. However, unlike JavaMOP, Tracematches maintains a separate indexing tree for each state, rather than multiple versions for different parameter sets. Consequently, it may be necessary to sequentially iterate over all key–value pairs at certain levels of the map in some cases. These cases typically occur when the incoming event does not bind the variable situated at the top level of the map. To mitigate this, Tracematches employs heuristics to eliminate such situations as much as possible.

Algorithm 3.2.1 Pseudocode of the monitoring algorithm used in Tracematches (partially taken from [1]).

Input: A *trace* consisting of individual events.
Output: A set of parameter instances representing solutions.

```
1: constraints.init()        ▷ assigns true to the initial state, false to all remaining states
2: solutions.init()          ▷ initialize to empty set
3: for each event in trace do
4:     results.reset()
5:     for each symbol in symbols_matching(event) do
6:         bindings = get_parameter_bindings(event, symbol)
7:         for each (i, j) in transitions_with_symbol(symbol) do
8:             results[j] = results[j] ∨ get_increment_change(constraints[i], bindings)
9:         end for
10:    end for
11:    for each i in state_ids do
12:        constraints[i] = get_new_constraints(constraints[i], results[i])
13:        if i is final_state then
14:            solutions.append(get_new_solutions(constraints[i])
15:        end if
16:    end for
17: end for
18: return solutions
```

Tracematches also incorporates several forms of garbage collection. Naturally, it discards constraints associated with parameter bindings that have already matched the property. In addition, it removes constraints tied to parameter bindings that can no longer possibly lead to a match. Since parameter bindings correspond to objects from the monitored program, such situations can occur, for example, when an object is garbage collected by the JVM, yet the binding would require future events involving that object to complete a match. In such cases, it is entirely safe to discard the related constraints. The tool employs various heuristics to determine whether a particular binding still has the potential to satisfy the property.

3.3 Larva

Larva [6] is another popular tool based on the *trace slicing* approach. Unlike the previously mentioned tools, it is primarily focused on monitoring real-time properties. This focus is also reflected in its internal implementation which is based on *Dynamic Automata with Timers and Events* (*DATE*). A *DATE* is essentially a wrapper for a collection of monitor instances associated with a particular property. Larva uses a custom variant of symbolic state automata to represent these monitor instances. This automaton includes timers (which can be stopped, resumed or reset) and may also contain a set of state variables. Each transition in the automaton is defined by:

- An event (either from the monitored system or triggered by a timer),
- a condition on state variables or timer values,

- an action (e.g., updating state variables or timer),
- an optional signal to other automata via dedicated channels.

DATE can be viewed as a dynamic network that organizes monitor instances into a hierarchical structure and creates new instances as needed.

Properties can also be parameterized using quantifiers over sets of objects. As a result, Larva instantiates a new monitor for each unique combination of relevant objects. Quantifiers can even be nested, resulting in a hierarchical arrangement of monitor instances within the *DATE* structure. Each child instance has access to the state variables of its parent. To determine whether a monitor instance for a given object already exists, Larva relies on equality by value.

An example of a base *DATE* automaton expressing the property *no more than two successive failed login attempts are allowed and the user may not be inactive for longer than 30 minutes* is shown in Figure 3.3. This property includes two distinct negative scenarios, and hence, two separate final states. Each transition is labeled using the following format—`event name / condition on state variables or timers (optional) / action to perform (optional)`.

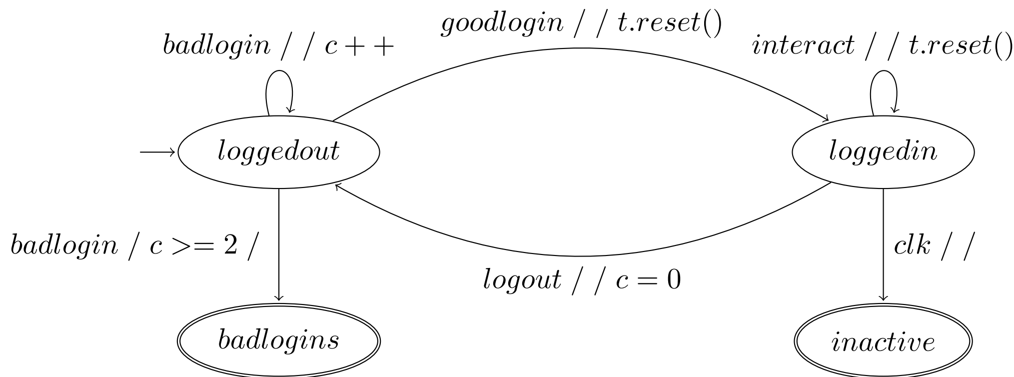


Figure 3.3: Example of the base automaton for *DATE* covering the *bad logins* property (partially retrieved from [6]).

Properties are typically specified in the form of a Larva script. This script is a direct textual representation of the *DATE* to be used for monitoring. However, for user convenience, Larva also supports different specification languages, e.g., Quantified Discrete-time Duration Calculus (QDDC) or Lustre (a synchronous dataflow programming language). All of them are translated into a corresponding *DATE* during the initialization phase. Listing 3.3 shows an example of a Larva script that corresponds to the *DATE* from Figure 3.3. The script consists of several sections. The first one is the definition of local variables to be used in a property. Then there is a set of events which trigger state transitions. Finally, there is also a textual representation of the automaton for the property.

```
GLOBAL {
    VARIABLES {
        int c = 0;
        Clock t;
    }
    EVENTS {
```

```
        badlogin() {*.badlogin()}
        goodlogin() {*.goodlogin()}
        interact() {*.interact()}
        logout() {*.logout()}
        clk() {t@30*60}
    }
    PROPERTY users {
        STATES {
            BAD { badlogins inactive }
            NORMAL { loggedin }
            STARTING { loggedout }
        }
        TRANSITIONS {
            loggedout -> badlogins [badlogin \ c>=2 \ ]
            loggedout -> loggedin [goodlogin \ \ t.reset();]
            loggedout -> loggedout [badlogin \ \ c++;]
            loggedin -> inactive [clk \ \ ]
            loggedin -> loggedout [logout \ \ c=0;]
            loggedin -> loggedin [interact \ \ t.reset();]
        }
    }
}
```

Listing 3.3: Example of the Larva script (partially retrieved from [6]).

The specification in Listing 3.3 does not employ any quantifiers. A property can be quantified by enclosing it in the `FOREACH` construct instead of the `GLOBAL` construct. `FOREACH` constructs can even be nested within each other, thus adding more parameters to the property. An example of such nesting is demonstrated in Listing 3.4.

```
FOREACH (User u) {
    FOREACH (Account a) {
        Property p {
            ...
        }
    }
}
```

Listing 3.4: Example of the quantification in Larva (partially retrieved from [5]).

Larva employs very basic garbage collection of monitor instances. An existing monitor instance is discarded only once it reaches an accepting state. This design choice is mainly based on the assumption that data objects associated with a particular monitor instance can be serialized and later deserialized during a monitoring session. Therefore, a monitor instance cannot be discarded even after the associated objects are no longer active in the monitored program. On the other hand, this means that Larva could accumulate a high number of monitor instances that are rarely updated and very likely irrelevant. This can lead to constantly increasing memory consumption.

Chapter 4

Plogchecker 2.0

As mentioned earlier, the main objective of this thesis is the reimplementation and improvement of the *Plogchecker* tool. This chapter provides details on the approaches and algorithms used in its latest version, *Plogchecker 2.0*. Section 4.1 offers a basic overview of the tool and explains its processing flow. Section 4.2 introduces the approach used for property specification. Section 4.3 discusses the details of the monitoring algorithm and lists its flaws. Finally, Section 4.4 focuses on the approaches used for garbage collection of monitor instances.

4.1 Overview

Plogchecker 2.0 [24] is a tool for *parametric runtime monitoring* that leverages the *trace slicing* approach. It is implemented in the Go programming language. This version is a reimplementation of the original *Plogchecker* [21], whose proof-of-concept was developed in Python. Unlike other runtime monitoring tools, *Plogchecker 2.0* is specifically designed to monitor properties over log records. In other words, it does not require instrumenting the monitored system with code to produce individual events. Instead, it parses events directly from the log records naturally generated by systems. This approach provides significant flexibility, as it theoretically supports all log formats. The tool supports both online monitoring (reading logs from a running system via standard input) and offline monitoring (processing logs stored in specified files).

The tool requires all information regarding the monitoring process to be specified in YAML configuration file. This file defines the properties to be monitored, along with the events and their parameters. For each event, a pattern is provided to extract the event from the corresponding log record. A more detailed explanation of the file format is available in Section 4.2. *Plogchecker 2.0* supports two types of properties:

- **Good property**: defines the desired behavior of the system. This property is violated when the sequence of events observed during monitoring does not conform to it.
- **Bad property**: describes undesired behavior of the system. This property is violated when the observed sequence of events matches it.

The processing flow of *Plogchecker 2.0* is illustrated in Figure 4.1. The tool operates in four distinct phases:

1. **Configuration Parsing**: The tool begins by parsing the input YAML configuration file. It extracts the definitions of properties and generates corresponding monitor instances. Additionally, it retrieves patterns for all events to enable event extraction from the log file.

2. **Event Extraction**: The filter component processes the input log file (or reads from standard input in online mode). Using the patterns obtained in the previous phase, it extracts events from the log records. These events are then passed continuously to the monitoring algorithm component for further processing.

3. **Monitoring process**: The monitoring algorithm component processes the incoming events, updates all monitor instances accordingly and optionally creates new instances based on parametric events. It also tracks the verdict categories for each monitor instance to provide a summary of violations.

4. **Report Generation**: After processing all log records, the monitoring algorithm component produces a final report in JSON format. This report contains details of all violations encountered during monitoring.

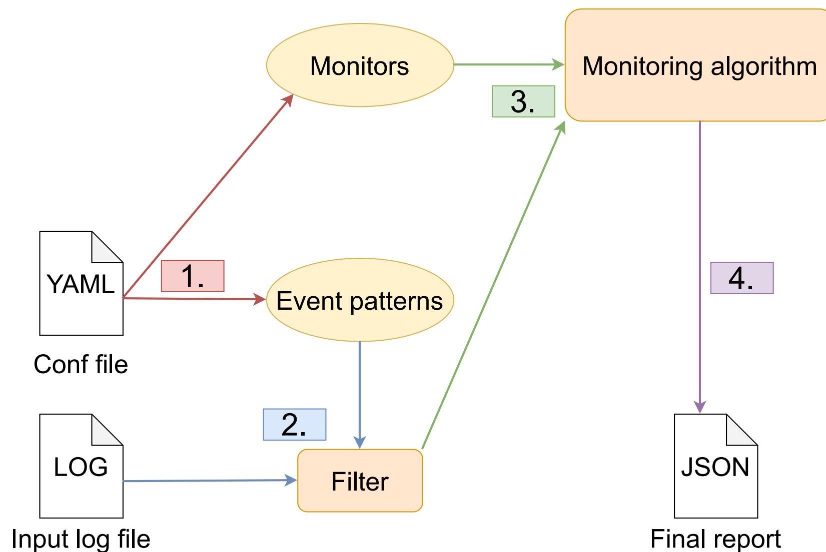


Figure 4.1: Data processing flow in the *Plogchecker 2.0*.

4.2 Property Specification

When using *Plogchecker 2.0*, all data related to the monitoring process must be specified in a special YAML configuration file at launch. This file defines the properties to be monitored and describes how to extract events from the input log records. An example configuration file is shown in Listing 4.1 which specifies a single property—the *unsafe iterator* property (as defined in Section 3.1). The configuration file consists of up to four sections:

properties* and *bad_properties
: The `properties` section is dedicated to good properties, while the `bad_properties` section specifies bad properties. Both sections are optional, but at least one property (good or bad) must always be defined. Each property is expressed as a RE composed of events. Every referenced event must be defined in the `events` section.

events
: Each event is defined by a pattern, represented as a RE, to facilitate event extraction from log records. Each event pattern can also include parameters associated with events. Several basic data types are supported for these parameters.

constraints
: This section defines relationships and restrictions, either between event parameters or between event parameters and constant expressions.

```
bad_properties:
  UnsafeIterator: CreateI Next* UpdateC+ Next
events:
  CreateI: "create(%{WORD:c}) -> %{WORD:i}"
  UpdateC: "udate(%{WORD:c})"
  Next: "next(%{WORD:i})"
constraints:
  - 'CreateI.c = UpdateC.c'
  - 'CreateI.i = Next.i'
```

Listing 4.1: Example of configuration file for *Plogchecker 2.0*.

Properties

The definitions of both good and bad properties must be provided as RE composed of events. Each property is described by an event identifier and its corresponding RE. For example, Listing 4.1 illustrates the definition of a single bad property, *UnsafeIterator*.

Plogchecker 2.0 currently supports the following operators in RE for property definitions:

- Concatenation (`" "`),
- alternation (`|`),
- iteration (`*`),
- positive iteration (`+`),
- bounded iteration (`{n}` or `{n1,n2}`).

Additionally, the RE defining a bad property can include the discard operator `"!"`. This operator is used to indicate that a particular bad property can no longer be matched if a specific sequence of events is encountered during the monitoring process. For example, consider a property defined by the following RE (assuming it is non-parametric for simplicity): `p1: a+ c! b`. This specification means that if a `"c"` event occurs after an initial sequence of `"a"` events, the property `p1` can no longer be matched. For illustration, consider the trace `"a a c b"`. In this case, the bad property `p1` will not be matched because the `"c"` event invalidates further matches of the property.

Events

As previously mentioned, each event is defined by a pattern in the form of a RE, which can optionally include parameter definitions. Each parameter must be assigned a specific data type. The data type specification is implemented using type patterns from the Logstash filter plugin, `Grok`[1]. Consequently, each event parameter in the *Plogchecker 2.0* configuration file is specified using the following format:

$$\%< DATA_TYPE >:< PARAMETER_NAME >$$

For example, Listing 4.1 demonstrates several parameters of type `WORD`. *Plogchecker 2.0* currently supports the following data types (with sample values provided for clarity):

- `NUMBER` (e.g., `32` or `-23`),
- `WORD` (e.g, `foo` or `baz_42`),
- `DATESTAMP_RFC1123` (e.g, `Sun, 09 Feb 1997 15:34:42 GMT`),
- `DATE_ISO8601` (e.g, `2006-02-09T15:34:42`).

Constraints

To enhance flexibility, *Plogchecker 2.0* enables the specification of constraints for parameters used in events. These constraints are defined as simple expressions where the operands can be either event parameters or constants. All expressions must adhere to the data types defined for the parameters. Additionally, two extra data types, `DURATION` and `BOOL`, are supported for constants to provide greater convenience.

Currently, the following operators are supported in expressions:

- Equal (`=`),
- not equal (`!=`),
- greater than (`>`),
- less than (`<`),
- greater or equal than (`>=’`),
- less or equal than (`’<=’`),
- addition (`+`),
- subtraction (`-`).

Additionally, expression can also contain following two functions:

- `is_substr(s1: WORD, s2: WORD): BOOL`: checks if `s2` is substring of `s1` .
- `length(s: WORD): NUMBER`: retrieves the length of the `s`.

[1]Grok filter plugin—https://www.elastic.co/docs/reference/logstash/plugins/plugins-filters-grok.

Operators and functions cannot be applied to operands arbitrarily. Instead, their usage must adhere to type-specific semantic rules. These rules ensure that expressions can be correctly evaluated during the monitoring process. To enforce this, *Plogchecker 2.0* performs semantic checks while parsing the configuration YAML file and constructing the monitors.

4.3 Monitoring Algorithm

Plogchecker 2.0 uses finite-state machines (FSMs) as the foundational formalism for its monitors. However, it employs an unconventional approach for their internal representation, i.e., the use of custom table structures. The main aim of this thesis is to address this by redesigning and refactoring the monitoring algorithm, as the current implementation is difficult to comprehend and maintain. More importantly, it exhibits incorrect behavior in certain corner cases.

For illustration, consider the unsafe iterator property specified in Listing 4.1. The FSM corresponding to this property is depicted in Figure 4.2. This automaton features four highlighted transitions, each of which is treated as unique by the monitoring algorithm in *Plogchecker 2.0*. For every such transition, the algorithm maintains a dedicated table structure.

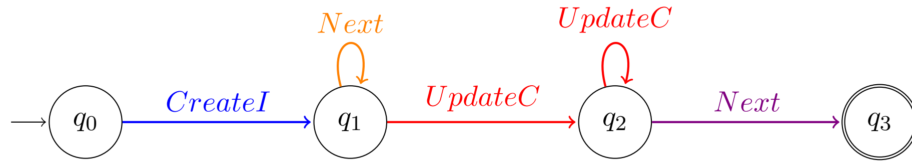


Figure 4.2: FSM corresponding to the *unsafe iterator* property where transitions are highlighted according to philosophy in *Plogchecker 2.0*.

During event processing, the algorithm populates the tables incrementally. Each row in a table corresponds to a parameter instance from a received event relevant to that table. These parameter instances do not need to be unique across rows. Furthermore, each row can reference other rows, enabling the algorithm to track the sequence of received events and represent the state of each monitor. Figure 4.3 illustrates these tables and their evolution while processing the sample trace.

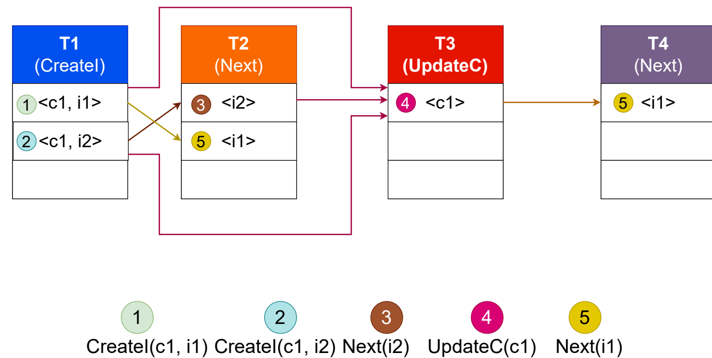


Figure 4.3: Demonstration of table structure usage in *Plogchecker 2.0* when monitoring *Unsafe iterator* property.

Plogchecker 2.0 distinguishes between event sequences even within the same parameter trace. For example, for the trace `"createI(c1, i2) next(i2) UpdateC(c1)"`, it will internally create several event sequences (e.g., `"createI(c1, i2) next(i2) UpdateC(c1)"` and `"createI(c1, i2) UpdateC(c1)"`). This approach deviates from the traditional interpretation of *trace slicing* (as defined in Definition 2.2.6). Instead of maintaining a single monitor instance for each parameter instance, *Plogchecker 2.0* creates multiple monitor instances per *trace slice*, one for each unique event sequence. Since these instances contain identical information, if one matches the property, all will. Therefore, this approach results in unnecessary resource consumption. Moreover, it prevents the use of popular indexing techniques that rely on parameter instances.

The simplified pseudocode for the monitoring algorithm is presented in Algorithm 4.3.1. The algorithm processes events sequentially from the input trace. For each event, it iterates over groups of tables organized by property. From each group, only the tables relevant to the current event (i.e., those dedicated to transitions involving this event) are selected for further processing. For each relevant table t_e, the algorithm first verifies if the constraints specified in the configuration file are satisfied. If the constraints are met, it retrieves a list of starting tables—those from which references to t_e can theoretically be created. It then determines whether new references can be established from any of these starting tables in compliance with the processed event, creating such references as a side effect. Finally, if the event is a creation event or if new references were successfully established, the parameter instances from the event are added to t_e.

Algorithm 4.3.1 Pseudocode of the monitoring algorithm utilized in *Plogchecker 2.0* (partially taken from [24]).

Input: A parametric trace $\tau \in E(X)^*$.
Globals: Tables grouped by individual properties $T_{property_groups}$.
1: **for each** $e(\theta)$ **in** τ **do**
2:   **for each** $T_{property}$ **in** $T_{property_groups}$ **do**
3:     **for each** t_e **in** $get_relevant_tables(T_{property}, e(\theta))$ **do**
4:       **if** $are_constraints_satisfied(t_e, e(\theta))$ **then**
5:         $connected = false$
6:         **for each** t **in** $get_starting_tables(t_e, e(\theta))$ **do**
7:           $items = get_relevant_items(t, e(\theta))$
8:           $connected = connected \lor check_if_connected(t, items, e(\theta), t_e)$
9:         **end for**
10:         **if** $connected \lor is_creation_event_table(t_e)$ **then**
11:           $add_to_table(t_e, e(\theta))$
12:         **end if**
13:       **end if**
14:     **end for**
15:   **end for**
16: **end for**

4.4 Garbage Collecting

Since the monitoring process can be memory-intensive, *Plogchecker 2.0* employs a basic form of garbage collection to maintain reasonable memory usage. In its current imple-

mentation, the tool discards only closed event sequences as part of this garbage collection process. To clarify what is meant by closed event sequences, consider a property defined by the following RE: `A B (C* D)*`. Assume this property is parameterized by a single parameter, x, which is included in all four events. A sample trace for this property, along with its corresponding representation in the internal table structures, is illustrated in Figure 4.4. The tool uses four tables in total, two of which (`T2` and `T4`) are marked as final because they contain items representing the last events in sequences that match the RE. Consequently, all sequences ending with items from these tables are considered closed. The categorization of event sequences is demonstrated in the bottom part of the figure.

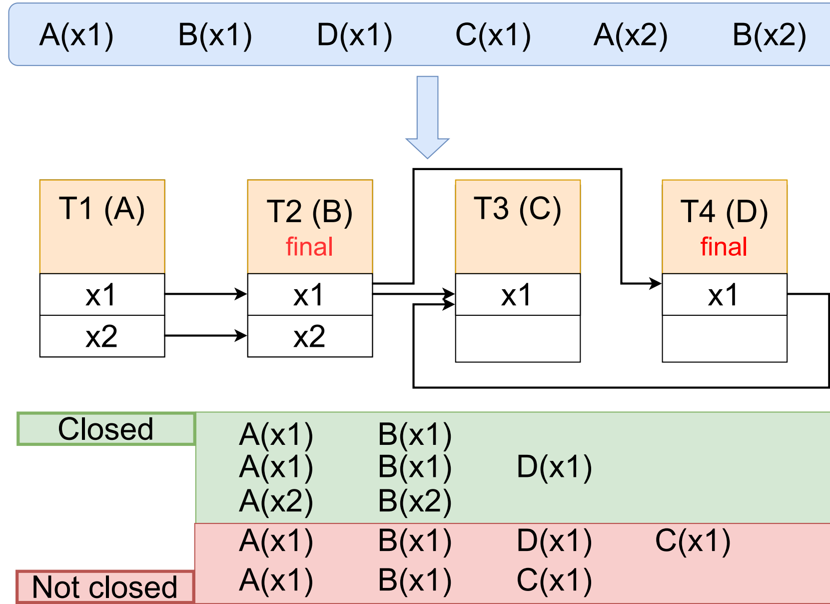


Figure 4.4: Demonstration of the *closed event sequences* for property `A B (C* D)*` with single parameter (partially adopted from [24]).

Since the garbage collection process in *Plogchecker 2.0* can discard all closed event sequences, it has an important implication. For bad properties, only the minimal sequences of events (i.e., the shortest prefixes) are guaranteed to be detected as violations. This behavior is illustrated by the `"A(x2) B(x2)"` event sequence in Figure 4.4. Because this sequence is closed, it may be garbage collected. Consequently, if a `"D(x2)"` event is received later, the violation for the sequence `"A(x2) B(x2) D(x2)"` will not be detected.

The garbage collection mechanism in *Plogchecker 2.0* is implemented using two threads. One for the monitoring algorithm and the other for the garbage collection process. These threads are synchronized using a mutex lock which protects the critical section represented by the set of tables used for monitor representation. A schematic illustrating the roles of each thread is provided in Figure 4.5. The procedures performed by the threads are as follows:

Monitoring algorithm thread initially waits for an incoming event to process. Once an event is received, it attempts to acquire the lock. After successfully obtaining the lock, it processes the event and updates the internal tables (as described in Sec-

tion 4.3.1). The lock is then released, and the thread returns to its initial state, waiting for the next event.

Garbage collecting thread begins by sleeping for a duration of `X` seconds. The value of `X` is currently hardcoded in the tool's implementation and set to 5 seconds. After waking up, the thread attempts to acquire the lock. Once the lock is obtained, it sequentially examines all tables to identify those where the number of items exceeds a threshold `Z`. The value of `Z` is also hardcoded and currently set to 100. For each identified table, the thread removes all items corresponding to closed sequences. The lock is then released, and the thread repeats the process, starting with another sleep cycle of `X` seconds.

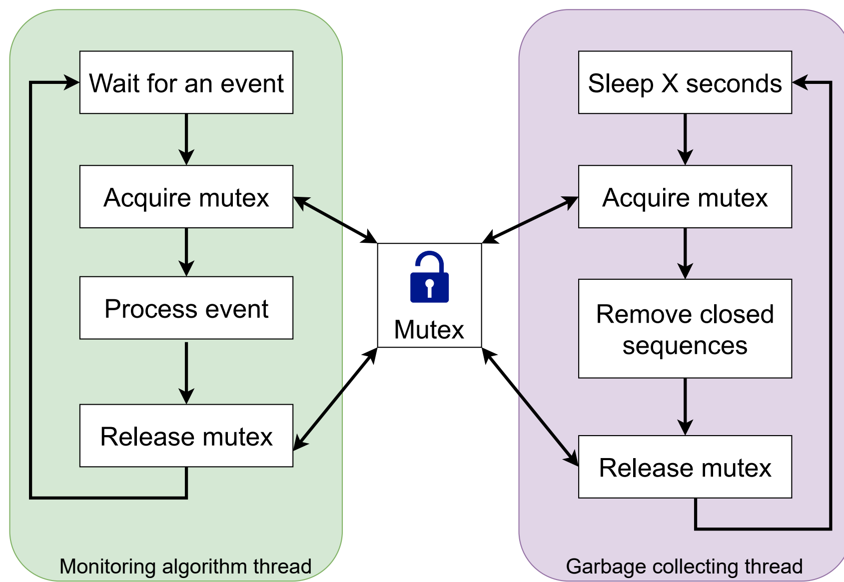


Figure 4.5: Scheme of the cooperation between the monitoring algorithm and the garbage collection (partially adopted from [24]).

Chapter 5

Enhancements Proposal

This chapter describes the design proposal for *Plogchecker 3.0*, a reimplementation and enhanced version of its predecessor—*Plogchecker 2.0* (see Chapter 4 for details). The requirements for the new version are outlined in Section 5.1. A general overview of the new tool is provided in Section 5.2. Subsequently, Section 5.3 discusses the approach to processing property specifications and explains the proposed enhancements. As noted in Section 4.3, the monitoring algorithm in *Plogchecker 2.0* has limitations. Therefore, a new algorithm has been proposed, with a detailed explanation available in Section 5.4. This algorithm supports multiple monitoring modes, which are further discussed in Section 5.5. Finally, Section 5.6 presents proposed approaches for efficient resource management and release.

5.1 Requirements

This section briefly defines the requirements for *Plogchecker 3.0*. The list of non-functional requirements is provided in Table 5.1.

ID	Description
1.1	Monitoring of multiple properties simultaneously must be supported (scalability with respect to the number of properties).
1.2	Monitoring of multiple independent parameters within properties must be supported (scalability with respect to the number of parameters).
1.3	Cross-platform compatibility must be ensured (e.g., Windows, Linux, macOS).
1.4	Users must be able to specify the maximum memory usage limit for the tool at launch.
1.5	Resources must be preemptively released based on a user-selected strategy (e.g., LRU, LFU).

Table 5.1: The non-functional requirements for the *Plogchecker 3.0*

Additionally, Table 5.2 lists the functional requirements for *Plogchecker 3.0*.

5.2 New Design Overview

This section provides a brief overview of *Plogchecker 3.0*. As with its predecessor, Go has been chosen as the primary implementation language. This choice is due to Go's excellent

ID	Description
2.1	The tool processes a sequence of events as its input, where each event is a tuple in the format '(<event_id> <param$_1$> ... <param$_n$>)'.
2.2	Online monitoring is supported (i.e., events are generated live during the tool's execution).
2.3	Offline monitoring is supported (i.e., events were produced in advance and captured into a file).
2.4	Users can request the monitoring of non-parametric properties specified using RE consisting of events without parameters.
2.5	Users can request the monitoring of parametric properties specified using RE consisting of events, where each event can carry one or more parameters.
2.6	Users can define both good_properties (i.e., desired system behavior) and bad_properties (i.e., undesired system behavior).
2.7	Each parameter is assigned a data type chosen by the user (e.g., STRING, NUMBER, DATE, IP, or PATH).
2.8	Users can provide expressions to describe constraints between parameters.
2.9	Constraint expressions can include basic arithmetic and relational operators.
2.10	Constraint expressions can also include a selected set of function operators (e.g., is_parent_directory(), is_substring(), length()).
2.11	Constraint expressions can include literals of various data types (e.g., STRING, NUMBER, DATE, IP, PATH, DURATION, or BOOL).
2.12	Users can use the discard operator '!' in property specifications to indicate that if the specified sequence of events is received, the property can no longer be matched. User must be able to use it both good and bad properties.

Table 5.2: The functional requirements for the *Plogchecker 3.0*.

balance between developer-friendly syntax and high performance. Additionally, it offers straightforward cross-platform compatibility (Requirement 1.3).

The overall behavior of the tool is similar to the previous version. The most significant change of the requirements is that *Plogchecker 3.0* no longer parses events from a log file. Instead, it expects a sequence of events as input (Requirement 2.1). Each input event is represented as a tuple, with the first item being the identifier of the event and the remaining items representing the event's parameters. As a result, the size of the tuple can vary across different events. Input events may originate from the live system during monitoring (i.e., online monitoring—Requirement 2.2), or be read from a pre-recorded file (i.e., offline monitoring—Requirement 2.3).

The data flow in *Plogchecker 3.0* is illustrated in Figure 5.1. Unlike its predecessor (Section 4.1), *Plogchecker 3.0* operates in three phases. Notably, the phase that involved extracting events from a log file has been removed. Instead, event extraction is now handled by external helper tools. This change is driven by the principle of separation of concerns, as *Plogchecker 2.0* previously handled both event extraction and property monitoring. As a result, *Plogchecker 3.0* can now be fully focused and optimized on its main objective—monitoring property violations. The phases are as follows:

1. **Configuration parsing**: This phase is largely the same as in the predecessor. The only difference is that *Plogchecker 3.0* no longer needs to extract events from a log file. As a result, this phase now only involves loading property specifications, validat-

ing constraints, and generating the base monitor for each property. A more detailed explanation of the actions in this phase can be found in Section 5.3.

2. **Monitoring process**: The second phase is where the actual monitoring occurs. It continuously processes input events (represented as tuples). Each event is processed by the newly proposed monitoring algorithm which is described in detail in Section 5.4. In addition to the monitoring algorithm, several helper threads are used to ensure that resource usage remains within reasonable bounds. More information on the proposed helper threads is presented in Section 5.6.

3. **Report Generation**: No major changes have been made to this phase. Once all events in the input sequence are processed, *Plogchecker 3.0* generates a JSON report containing all observed property violations. Additionally, similar to its predecessor, it supports continuous notifications about violations even before the monitoring process is complete. This applies only in cases where property violations can be determined before processing the entire event sequence.

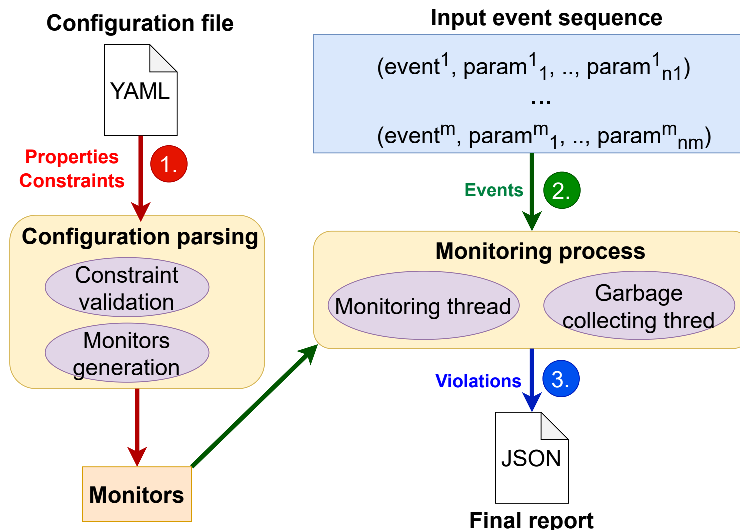


Figure 5.1: Overview of the data flow in *Plogchecker 3.0*.

5.3 Property Specification

The specification of properties to be monitored is a crucial aspect of any *runtime verification* tool. In *Plogchecker 3.0*, this component remains largely unchanged from its predecessor. No major modifications have been introduced, apart from adding support for several new data types and operators to enhance the tool's expressiveness. Consequently, property specifications are still expected to be provided in the form of a YAML configuration file. This file can include the following sections:

- ***properties* and *bad_properties* sections**: define the properties to be monitored, encompassing both good and bad properties (Requirement 2.6). Each property is specified using a RE composed of event identifiers. If none of the events in the RE has

parameters, the property is considered non-parametric (Requirement 2.4). Otherwise, it is classified as a parametric property (Requirement 2.5).

- ***events* section**: specifies the events that can be used in the RE for properties. Event definitions may optionally include parameters, with each parameter assigned a data type (Requirement 2.7). While *Plogchecker 3.0* no longer handles event extraction from log files, the event definitions can still optionally include patterns for their extraction, in addition to their identifiers and parameters. This requirement ensures compatibility with external helper tools that may use the same YAML configuration file for event extraction.

- **constraints section**: defines expressions that represent relationships and constraints for parameters (Requirement 2.8). These expressions can involve both parameters and constant literals as operands (Requirement 2.11). Constraints can be specified using either various arithmetic or relational operators (Requirement 2.9), or selected functional operators (Requirement 2.10).

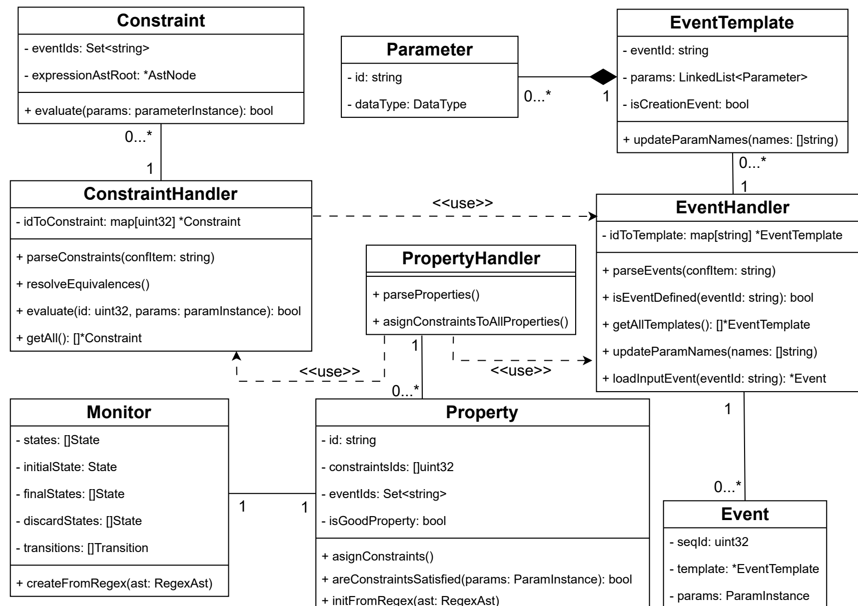


Figure 5.2: Class diagram covering configuration processing of the *Plogchecker 3.0*.

Structural Design of the Property Specification Processing

The structural design for implementing the YAML configuration processing is illustrated in Figure 5.2. Each distinct section in the YAML configuration file is managed by a dedicated handler:

- `PropertyHandler`: is responsible for parsing the *properties* and *bad_properties* sections and populating the corresponding internal representations.

- `EventHandler`: handles the parsing of the *events* section. Additionally, it is tasked with processing input events during monitoring, represented as tuples.

- `ConstraintHandler`: Manages the parsing of the *constraints* section and ensures that the internal representations are correctly populated.

Processing of *properties* and *events* Sections

Figure 5.3 presents an activity diagram that illustrates the flow of parsing and processing the *properties* and *events* sections in the configuration YAML file. The process begins with the definition of events. For each event, the event identifier is extracted. Additionally, if an event has parameters, these are also extracted, including their data types. All the extracted information is then loaded into the appropriate internal representation.

After processing all events, the procedure proceeds to the properties. The processing of both good and bad properties is identical, except for setting a flag that determines the property type. For each property, the first step is to tokenize the regular expression (RE) that specifies it. Subsequently, an abstract syntax tree (AST) is constructed from the tokens, if possible. If the construction fails, the configuration parsing terminates with a syntax error.

Once the AST is successfully created, it is used to convert the original RE into a corresponding NFA using the recursive procedure described in Algorithm 2.4.1. The resulting NFA is then transformed into a DFA by employing Algorithm 2.4.2. Finally, the DFA is minimized using Algorithm 2.4.3.

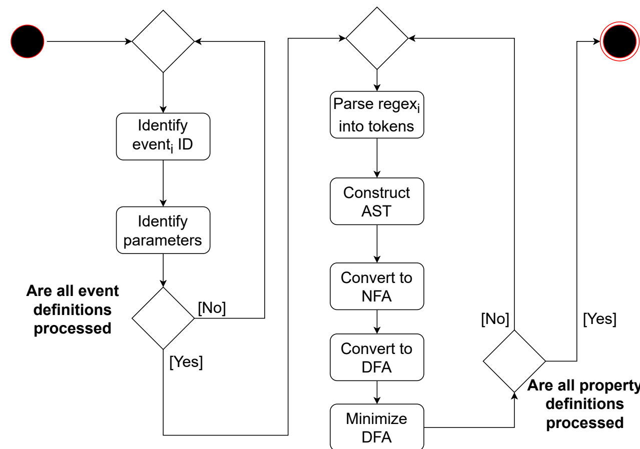


Figure 5.3: Activity diagram describing the processing of the property and event specifications in *Plogchecker 3.0*.

Processing of *constraints* Section

The processing flow for the *constraints* section is depicted in Figure 5.4. Each expression specifying a constraint is processed individually. Initially, the expression is tokenized into its constituent elements. These tokens are then used to construct an AST, provided the structure is valid.

Next, semantic checks are performed to ensure compliance with all rules, particularly those governing the application of operators to various data types. Once the semantic validation is complete, static optimizations are applied to the expression. This involves resolving branches in the AST that depend solely on literals rather than parameter values,

thereby reducing the AST's size as much as possible. The optimized AST is then stored as the representation of the constraint.

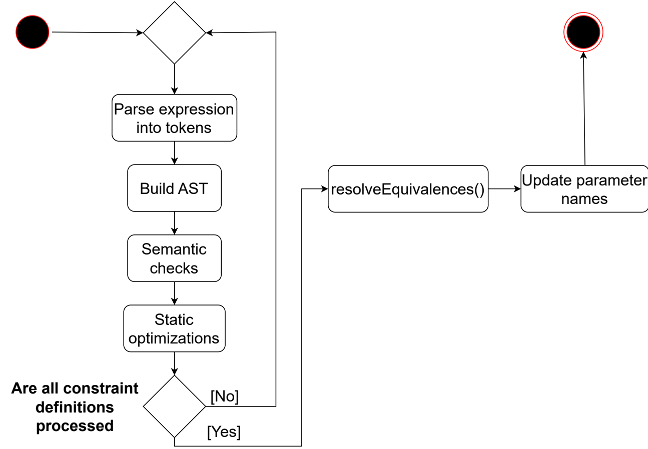


Figure 5.4: Activity diagram describing the parsing of the constraints in *Plogchecker 3.0*.

After all constraints have been processed, an additional optimization step is carried out. For the efficient monitoring, minimizing the number of distinct parameters is desirable. Therefore, the function `resolve_equivalences()` is employed to identify and resolve equivalences between parameters across constraints, effectively reducing their total number. The final step involves propagating these parameter adjustments to the corresponding event definitions.

The complete pseudocode for the `resolve_equivalences()` function is provided in Algorithm 5.3.1. This function aims to reduce the number of parameters by identifying equivalences between parameters across constraints. This task corresponds to the *Disjoint Set Union problem* (also known as the *Union-Find problem*) [10]. Accordingly, the pseudocode leverages a specialized data structure for this purpose, referred to as `union_find`. This data structure supports two fundamental operations:

- `find(x)`: Retrieves the root of the set to which the item `x` belongs.
- `union(x, y)`: Merges the sets containing `x` and `y`.

At the start of the algorithm, all parameters are treated as unique, each belonging to its own set. The constraints are then processed sequentially. If a constraint represents an equivalence (i.e., an expression with the `'='` operator) between two parameters, the union operation is applied to merge the sets of these parameters. Such constraints are removed from the list, as they have been statically evaluated and are no longer relevant.

Once all constraints have been processed, the algorithm constructs a mapping from the original parameter names to their new representations, reflecting the resolved equivalences.

5.4 New Design of the Monitoring Algorithm

As discussed in Section 4.3, the monitoring algorithm in *Plogchecker 2.0* has certain limitations. To address these and meet Requirements 1.1 and 1.2, a new algorithm has been de-

Algorithm 5.3.1 Algorithm for resolving parameter equivalences based on constraint analysis.

Input: A ist of original parameter names L_p.
Output: A mapping M of original parameter name to the new names.
Globals: A list of constraints C.

```
1:  function RESOLVE_EQUIVALENCES(L_p)
2:      union_find = init_union_find(L_n)
3:      C' = []
4:      M = {}
5:      for each c in C do
6:          if is_equivalence_for_two_params(c) then
7:              union_find.union(c.left, c.right)
8:          else
9:              C'.add(c)
10:         end if
11:     end for
12:     update_constraints(C')
13:     for each l_p in L_p do
14:         M[l_p] = union_find(l_p)
15:     end for
16:     return M
17: end function
```

signed for *Plogchecker 3.0*. This algorithm adopts a more traditional approach. For each received parameter instance, a corresponding monitor instance is created. Each monitor instance maintains the current state of the parameter instance within the associated base monitor automaton. The base monitor represents the monitored property. This design enables greater utilization of indexing and other optimizations commonly employed in state-of-the-art tools, such as JavaMOP (Section 3.1).

Indexing Technique

Plogchecker 3.0 employs indexing trees to optimize the retrieval of monitor instances associated with specific parameter instances. These trees are similar to those used in other parametric monitoring tools (Chapter 3). The basic structure of the proposed indexing technique is illustrated in Figure 5.5. All components in the scheme (e.g., indexing trees, pools) are property-specific. Multiple indexing trees are created, each dedicated to a distinct subset of parameters. Parameter value hashes serve as keys in these trees, rather than the values themselves, to prevent redundant storage of identical parameter values and to avoid repeated hashing during a single step of the monitoring algorithm.

The tree values represent either specific monitor instances (when the tree handles full parameter bindings for the property) or collections of instances (for partial bindings). To further enhance performance, each tree includes a dedicated cache that stores the most recently accessed instances. Instead of storing instances directly, the trees store their hashes. This design facilitates lazy cleaning of the indexing trees. When an instance is no longer needed, it is deleted from the pool but remains in the corresponding indexing trees until

the trees later detect and automatically remove broken mappings. This behavior is akin to weak references, as seen in Java.

Additionally, each property includes a single Parameter Global Table, which uniformly stores records of all parameter values under their hashes. The table also contains metadata, such as the number of monitor instances associated with each parameter value. This metadata supports the removal of entire branches in the indexing trees (Section 5.6).

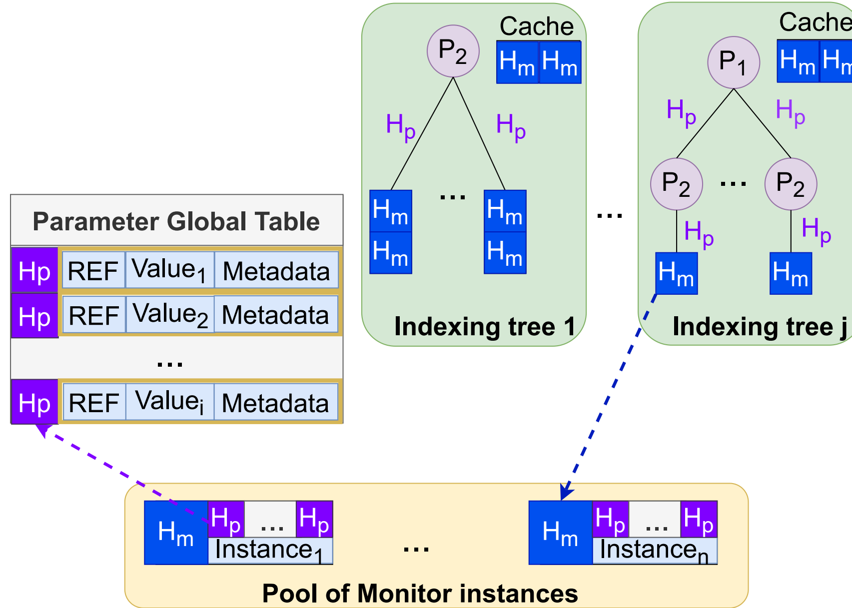


Figure 5.5: Scheme of the proposed indexing technique.

Multiple indexing trees are created for each property to ensure efficient lookups. To determine which parameter combinations require indexing trees, the procedure outlined in Algorithm 5.4.1 can be used. This procedure iterates over all events relevant to the property for which the indexing trees are being created. To optimize lookups for incoming events, the algorithm includes the parameter combinations present in each event into the resulting set. Additionally, the monitoring process must identify monitor instances from which new instances can be created by merging the parameter instance from the event with those in existing monitors. Consequently, the algorithm also incorporates any intersections of parameters between the event and other events into the resulting set.

Monitoring Algorithm

A brand-new monitoring algorithm has been designed for *Plogchecker 3.0* to address the limitations of its predecessor. The algorithm is proposed in a generic manner and supports multiple *monitoring modes*, which may slightly alter its behavior. These modes are described in detail in Section 5.5. For now, assume that the *monitoring mode* (denoted as `m` throughout this section) provides a set of operations that the monitoring algorithm can utilize.

The designed algorithm is presented in the form of an activity diagram in Figure 5.6. It outlines the processing flow applied to each property individually when handling a single input event.

Algorithm 5.4.1 Algorithm to determine for which parameter subsets indexing tree must be created.

Input: Set of events used in the property p—$E_p = e_1, ..., e_n$.
Output: Set P including subsets of parameters for which indexing tree must be created.

1: $P = \{\}$
2: **for** i **in** $\{1, ..., n\}$ **do**
3: $\quad e = E_p[i]$
4: $\quad$ **if** $is_parametric_event(e) \land e.param_names \notin P$ **then**
5: $\quad\quad P = P \cup \{e.param_names\}$
6: $\quad$ **end if**
7: $\quad$ **for** j **in** $\{i + 1, ..., n\}$ **do**
8: $\quad\quad e' = E_p[j]$
9: $\quad\quad params = e.param_names \cap e'.param_names$
10: $\quad\quad$ **if** $params \notin P$ **then**
11: $\quad\quad\quad P = P \cup \{params\}$
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: **end for**

As the first step, the algorithm checks whether the parameters in the input event satisfy the constraints. If not, no further actions are taken. If the parameters are valid, the algorithm inspects the relevant indexing tree (and its cache) to determine whether any monitor instances associated with the parameter instances in the event already exist. If such instances exist, the algorithm proceeds directly to updating them.

If no instances are found, the algorithm attempts to create a new monitor instance for the parameter instance in the event. It first tries to locate an existing compatible monitor instance from which a new instance can be derived by copying (using the func-

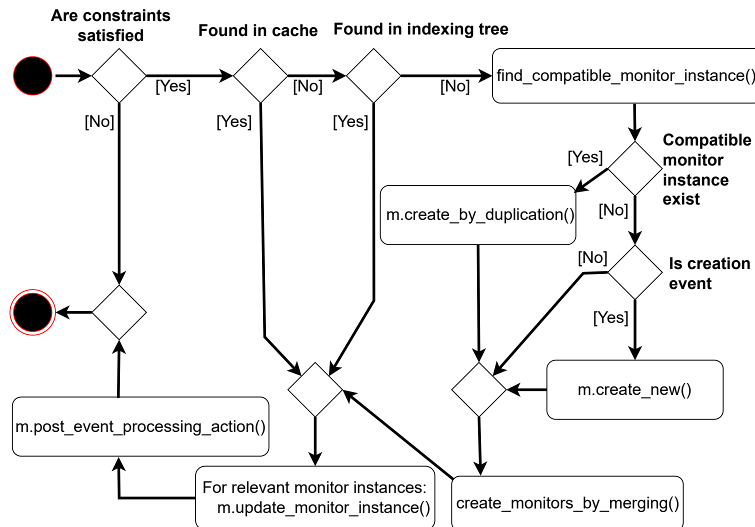


Figure 5.6: Activity diagram covering a proposed monitoring algorithm.

tion `find_compatible_monitor_instance()`, described below). If found, a new monitor instance is created using an operation provided by the *monitoring mode*.

If a monitor instance for the parameter instance in the event did not previously exist, the algorithm attempts to create additional instances by merging the parameter instance in the event with already existing monitor instances (via the function `create_monitor_by_merging()`, explained below).

Finally, all monitor instances for which the received event is relevant are updated. As the final step, the algorithm calls the post-processing hook defined by the *monitoring mode*.

The pseudocode for the function `find_compatible_monitor_instance()` is provided in Algorithm 5.4.2. Given a parameter instance θ, the function attempts to find an existing monitor instance associated with a parameter instance θ', such that θ' is compatible with the parameter bindings in θ and binds a smaller number of parameters.

The algorithm assumes that the indexing trees are sorted in descending order by the number of bound parameters. This guarantees that the monitor instance with the greatest overlap in parameter bindings is retrieved first. If no such instance exists, the function returns `undefined`.

Algorithm 5.4.2 Pseudocode of the function to find existing monitor instance compatible with the given parameter instance.

Input: A parameter instance θ for which compatible monitor existing monitor instance shall be found, a list T which includes all defined indexing trees sorted downwardly by the number of associated parameters.

Output: Monitor instance from which new instance for θ can be copied or $undefined$ if no such instance is found.

1: **function** FIND_COMPATIBLE_MONITOR_INSTANCE(θ, T)
2: $param_names = Dom(\theta)$
3: **for each** t **in** T **do**
4: **if** $t.param_names \not\subset param_names$ **then**
5: **continue**
6: **end if**
7: $\theta' = projection(\theta, t.param_names)$
8: $monitor = t.get_exact_match(\theta')$
9: **if** $monitor$ **is** $defined$ **then**
10: **return** $monitor$
11: **end if**
12: **end for**
13: **return** $undefined$
14: **end function**

Algorithm 5.4.3 presents the pseudocode for the function `create_monitor_by_merging()`. This function is responsible for creating new monitor instances by merging the given parameter instance θ with existing monitor instances.

First, it selects indexing trees whose associated parameter sets overlap with those in θ, but are not strict supersets of θ. This inclusion check ensures that only trees from which new monitor instances can actually be created are considered. To reduce unnecessary work, only trees with the smallest parameter sets are selected for each distinct overlap.

Next, for each selected tree, the algorithm searches for monitor instances that are compatible with θ. For each such compatible instance, it merges θ with the instance's parameter bindings to produce a new parameter instance θ_m. If θ_m satisfies the constraints and no monitor instance has yet been defined for it, a new monitor instance is created by using the operation provided by *monitoring mode*.

Algorithm 5.4.3 Pseudocode of the function that creates new monitor instances by merging the given parameter instance with existing monitor instances.

Input: A parameter instance θ to be used for merging, a list T containing all indexing trees sorted upwardly by the number of associated parameters, a *monitoring mode* m.
Output: List of the new monitor instances created by copying.

```
1: function CREATE_MONITORS_BY_MERGING(Dom(θ), T)
2:     M = {}
3:     for each t in GET_TREES_FOR_MERGING(Dom(θ), T) do
4:         θ′ = projection(θ, t.param_names)
5:         for each monitor in t.get_compatible_monitors(θ′) do
6:             θ_m = θ ⊔ monitor.get_params()
7:             if !is_instance_defined(θ_m, T) ∧ contraints_are_satisfied(θ_m) then
8:                 M.add(m.create_by_duplication(monitor, θ_m))
9:             end if
10:        end for
11:    end for
12:    return M
13: end function

1: function GET_TREES_FOR_MERGING(p, T)
2:     candidate_trees = {}
3:     for each t in T do
4:         p′ = t.param_names
5:         if (p ∩ p′! = ∅) ∧ (p ⊄ p′) then
6:             if !is_overlapping_tree_already_included(candidate_trees, t) then
7:                 candiate_trees.add(t)
8:             end if
9:         end if
10:    end for
11:    return candidate_trees
12: end function
```

Structural Design

The structural design for the implementation of the monitoring algorithm is provided by the class diagram in Figure 5.7. The main component is the `PropertyWatcher` class. An instance of this class is created for each property. `PropertyWatcher` is then associated with a single pool of monitor instances (represented by the `MonitorInstancePool` interface) to access individual monitor instances. Additionally, it may have up to one association with the `GlobalParamTable` to retrieve information

related to individual parameter values. Finally, it is also associated with several instances of the `IndexingTree` class to efficiently retrieve monitor instances.

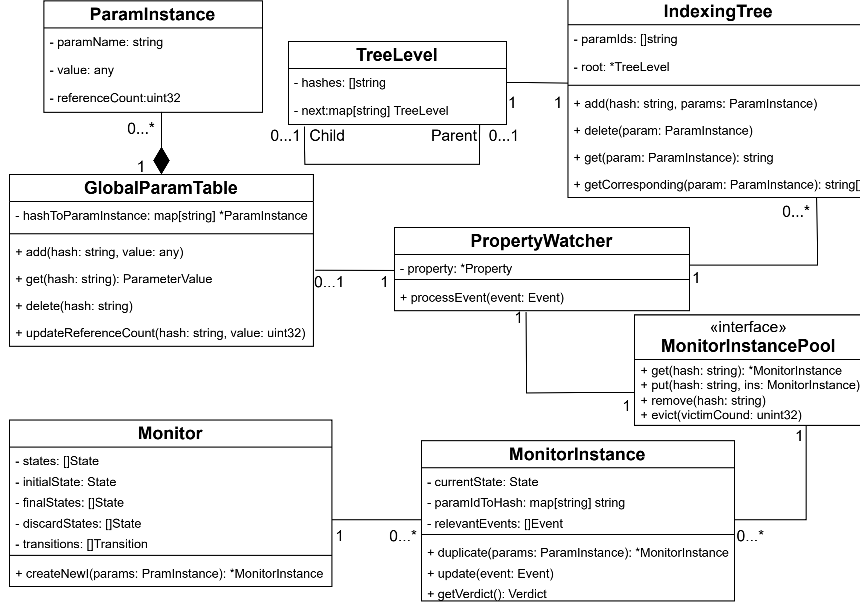


Figure 5.7: Class diagram illustrating the monitoring process architecture of *Plogchecker 3.0*.

5.5 Monitoring Modes

As presented in Section 5.4, the monitoring algorithm in *Plogchecker 3.0* is designed in a generic way. It can be easily parameterized by injecting a monitoring mode that provides basic operations over monitor instances. Before describing the individual monitoring modes, a common formal baseline is defined.

Definition 5.5.1 (Constraint over parameters)**.** Let X be a set of parameters and let V be a set of corresponding parameter values, as defined in Definition 2.2.3. Let the set of constraint verdicts be $O = \{satisfied, not_satisfied, unknown\}$. A **constraint over parameters** is defined as a total function $r : [X \rightharpoonup V] \to O$. The set of such constraints is denoted as $R \subseteq \{f \mid f : [X \rightharpoonup V] \to O\}$.

Definition 5.5.2 (Monitoring output function)**.** Let C_p be a set of output categories for a property, as defined in Definition 2.2.2. Let $D_p = (Q, \Sigma, \delta, q_0, F)$ be the DFA (Definition 2.4.2) representing the monitored property. The **monitoring output function** is a function that assigns an output category to each state: $\alpha : Q \to C_p$.

Definition 5.5.3 (Monitor instance)**.** Let $D_p = (Q, \Sigma, \delta, q_0, F)$ be the DFA representing a monitored property (Definition 2.4.2), let C_p be its set of output categories (Definition 2.2.2), and let R_p be the set of all constraints associated with this property (Definition 5.5.1). A **monitor instance** managing the parameter instance θ (Definition 2.2.3) is defined as a 5-tuple $m = (q, \theta, t, R, c)$ where:

- $q \in Q$ is the current state of the monitor instance in the base DFA,
- θ is the associated parameter instance,
- $t \in \mathbb{N}_0$ is the associated parameter instance (relevant for some of the monitoring modes),
- $R \subseteq R_p$ is the set of constraints associated with the monitor instance that are not yet satisfied,
- $c \in C_p$ is the output category associated with the monitor instance.

The set of all monitor instances for the property represented by D_p is denoted as M_p.

Definition 5.5.4 (Indexing tree over monitor instances)**.** Let X be a set of parameters and V a set of corresponding parameter values (Definition 2.2.3). Let $\mathcal{P}(A)$ denote the power set of a set A and let M_p be the set of monitor instances for a given property (Definition 5.5.3). Then, the **indexing tree over monitor instances** is defined as a pair $T = (T_1, T_2)$, where:

- $T_1 : [X \rightharpoonup V] \rightharpoonup M_p$ is a partial function mapping parameter instances to monitor instances,
- $T_2 : [X \rightharpoonup V] \to \mathcal{P}([X \rightharpoonup V])]]$ is a total function mapping a parameter instance to all strictly more informative parameter instances (i.e., the inverse of the relation from Definition 2.2.5).

Additionally, assume that T_2 is initialized to the empty set for all parameter instances: $\forall \theta \in [X \to V] : T2[\theta] = \emptyset$

Let θ be a parameter instance (Definition 2.2.3), let $e(\theta)$ be a parametric event and let m, m' be monitor instances (Definition 5.5.3). Then, a monitoring mode is characterized by the definition of the following set of operations:

- `create_new`(θ): creates a fresh monitor instance that manages the parameter instance θ.
- `create_by_duplication`(m', θ): creates a new monitor instance for managing the parameter instance θ by duplicating the existing parent monitor instance m'.
- `post_event_processing_action`$(e(\theta))$: defines an action to be performed after the input event $e(\theta)$ is fully processed.
- `update_monitor_instance`$(m, e(\theta))$: updates the monitor instance m using the input event $e(\theta)$.

Currently, *Plogchecker 3.0* supports two different monitoring modes, which differ in the strictness of event ordering. The supported modes are:

- *Standard*,
- *No Out-of-Order event* (*NOoOE*).

Standard Monitoring Mode

The *Standard* monitoring mode serves as the default mode in *Plogchecker 3.0*. In this mode, monitor instances are updated with relevant events only when a transition is possible, allowing events that arrive out of order to be silently skipped. For demonstration, see Listing 5.1, which shows a property definition together with a sample trace. The trace is considered matching in the *Standard* mode, even though the second event (i.e., e_3) arrived out of order—specifically, the monitor instance managing the parameter instance (p_1) was unable to make a transition with that event.

```
property:
    e1(p) e2(p) e(p)
trace:
    e1(1) e3(1) e2(1) e3(1)
```

Listing 5.1: Demonstration of the simple property together with the sample trace including the out of order event.

The complete definition of the operations characterizing this monitoring mode is provided in Algorithm 5.5.1.

When creating a fresh monitor instance for a parameter instance θ, it is assumed that no constraints are violated. A new monitor instance is therefore created and added to the indexing tree only if that condition is satisfied. This instance is initialized with all constraints whose verdicts cannot yet be determined (i.e., due to missing parameter values).

When creating a monitor instance by duplication, the process is largely the same, with the key difference being that the new instance is initialized using the state and data from the parent instance.

This mode does not require any special action to be performed after the event $e(\theta)$ has been processed.

During the update of a monitor instance with an event $e(\theta)$, it is first checked whether a transition exists to a non-dead state. If such a transition is available, the monitor instance's state is updated accordingly in the indexing tree.

No Out-of-Order Event Monitoring Mode

The behavior of this mode is inspired primarily by JavaMOP [19]. It enforces stricter rules when updating monitor instances—whenever an event relevant to some monitor instance is received, there must exist a transition from the current state to a non-dead state using that event. If such a transition does not exist, the monitor instance is considered non-matching for the input *trace slice* and is assigned the *violating* category. This enforcement is achieved internally by leveraging the concept of timestamps.

To illustrate the difference, consider the example from Listing 5.1. In the *NOoOE* mode, the monitor instance associated with parameter instance (p_1) will not produce a match due to the presence of an out-of-order event.

The definition of all operations characterizing this monitoring mode is provided in Algorithm 5.5.2. This algorithm introduces two additional global variables. The first, denoted as $\mathcal{T}$, is a mapping from parameter instances to the timestamp of the most recent event carrying that parameter instance. The second variable, τ, represents a global timestamp counter.

Algorithm 5.5.1 Definition of the operations in *Standard* monitoring mode.

Input: DFA representing the monitored property $D_p = (Q, \Sigma, \delta, q_0, F)$ (Definition 2.4.2), indexing tree $T = (T_1, T_2)$ (Definition 5.5.4), set of constraints for the property R_p (Definition 5.5.1) and monitoring output function $\alpha : s \to C$ (Definition 5.5.2).

1: **function** CREATE_NEW(θ)
2: **if** $\exists r \in R : r(\theta) = not_satisfied$ **then return end if**
3: $T_1(\theta) = (q_0, \theta, 0, \{r \in R \mid r(\theta) = unknown\}, don't_know)$
4: **for each** $\theta' \sqsubset \theta$ **do** $T_2(\theta') = T_2(\theta') \cup \{\theta\}$ **end for**
5: **end function**

1: **function** CREATE_BY_DUPLICATION((q, θ', t, R', c), θ)
2: **if** $\exists r \in R : r(\theta) = not_satisfied$ **then return end if**
3: $T_1(\theta) = (q, \theta, t, \{r \in R' \mid r(\theta) = unknown\}, c)$
4: **for each** $\theta' \sqsubset \theta$ **do** $T_2(\theta') = T_2(\theta') \cup \{\theta\}$ **end for**
5: **end function**

1: **function** POST_EVENT_PROCESSING_ACTION($e(\theta)$)
2: **end function**

1: **function** UPDATE_MONITOR_INSTANCE((q, θ, t, R, c), $e(\theta)$)
2: $q' = \delta(e)$
3: **if** $\forall w \in \Sigma^* : \delta^*(q', w) \notin F$ **then return end if**
4: $T_1(\theta) = (q', \theta, t, R, \alpha(q'))$
5: **end function**

When creating a fresh monitor instance for parameter instance θ, most steps mirror those in the *Standard* mode, except that the creation timestamp of the monitor instance is set to the current global timestamp τ, and τ is subsequently incremented.

When creating a new monitor instance by duplication, the mode performs an additional check to detect any previously seen out-of-order events. Specifically, it verifies whether there exists a parameter instance θ'' that is less informative than θ (the one carried by the currently processed event), not included in the parent instance θ', and has been observed in an event after the parent monitor instance was created. If such a parameter instance is found, creation of the new instance is aborted, as it would violate the out-of-order constraint.

After an event $e(\theta)$ is processed, the mode records the current global timestamp in $\mathcal{T}$ for the parameter instance θ and increments τ.

When updating a monitor instance with an event $e(\theta)$, the mode first checks whether a transition to a non-dead state exists for the current state. If no such transition is possible, the monitor instance is marked as *violating* and is excluded from further matching. All other aspects of processing follow the behavior defined in the *Standard* mode.

5.6 Improvements of the Garbage Collecting

Memory usage is often the primary bottleneck for parametric runtime verification tools. To address this challenge, *Plogchecker 3.0* has been designed with a focus on efficient memory management. The tool's processing flow consists of three threads, all of which are synchronized using an appropriate set of mutex locks. Since each thread needs to perform operations on properties, a dedicated lock is assigned to each property. This ensures

Algorithm 5.5.2 Definition of the functions in *NOoOE* monitoring mode.

Input: DFA representing the monitored property $D_p = (Q, \Sigma, \delta, q_0, F)$ (Definition 2.4.2), indexing tree $T = (T_1, T_2)$ (Definition 5.5.4), set of constraints for the property R_p (Definition 5.5.1) and monitoring output function $\alpha : s \to C$ (Definition 5.5.2).
Globals: mapping from parameter instances to the timestamp$\mathcal{T} : [[X \rightharpoonup V] \rightharpoonup \mathbb{N}_0]$, global timestamp counter $\tau \in \mathbb{N}_0$.
Init: $\tau = 0$

```
1: function CREATE_NEW(θ)
2:     if ∃r ∈ R : r(θ) = not_satisfied then return end if
3:     T1(θ) = (q0, θ, τ, {r ∈ R | r(θ) = unknown}, unknown)
4:     for each θ' ⊏ θ do T2(θ') = T2(θ') ∪ {θ} end for
5:     τ = τ + 1
6: end function

1: function CREATE_BY_DUPLICATION((q, θ', t, R', c), θ)
2:     if ∃r ∈ R : r(θ) = not_satisfied then return end if
3:     for each θ'' ⊑ θ do
4:         if θ'' ⊑ θ' then continue end if
5:         if 𝒯(θ'') > t then return end if
6:     end for
7:     T1(θ) = (q, θ, t, {r ∈ R' | r(θ) = unknown}, c)
8:     for each θ' ⊏ θ do T2(θ') = T2(θ') ∪ {θ} end for
9: end function

1: function POST_EVENT_PROCESSING_ACTION(e(θ))
2:     𝒯(θ) = τ
3:     τ = τ + 1
4: end function

1: function UPDATE_MONITOR_INSTANCE((q, θ, t, R', c), e(θ))
2:     q' = δ(e)
3:     if ∀w ∈ Σ* : δ*(q', w) ∉ F then
4:         T1(θ) = (q, θ, t, R', violating)
5:         return
6:     end if
7:     T1(θ) = (q', θ, t, R', α(q'))
8: end function
```

that a thread can modify data structures related to a specific property only after acquiring the corresponding lock.

The operations performed by each thread are illustrated in the activity diagram in Figure 5.8. A brief description of each thread is as follows:

Monitoring Thread

This thread is responsible for the core monitoring process. It waits for input events and adjusts all relevant properties accordingly. To operate efficiently, it uses a partially non-blocking mode. First, it identifies the pool of properties relevant to the received event. It then sequentially processes the properties in the pool. For each property, the thread attempts to acquire its dedicated lock. If the lock is obtained within a specified timeout, the property is processed, the lock is released, and the property

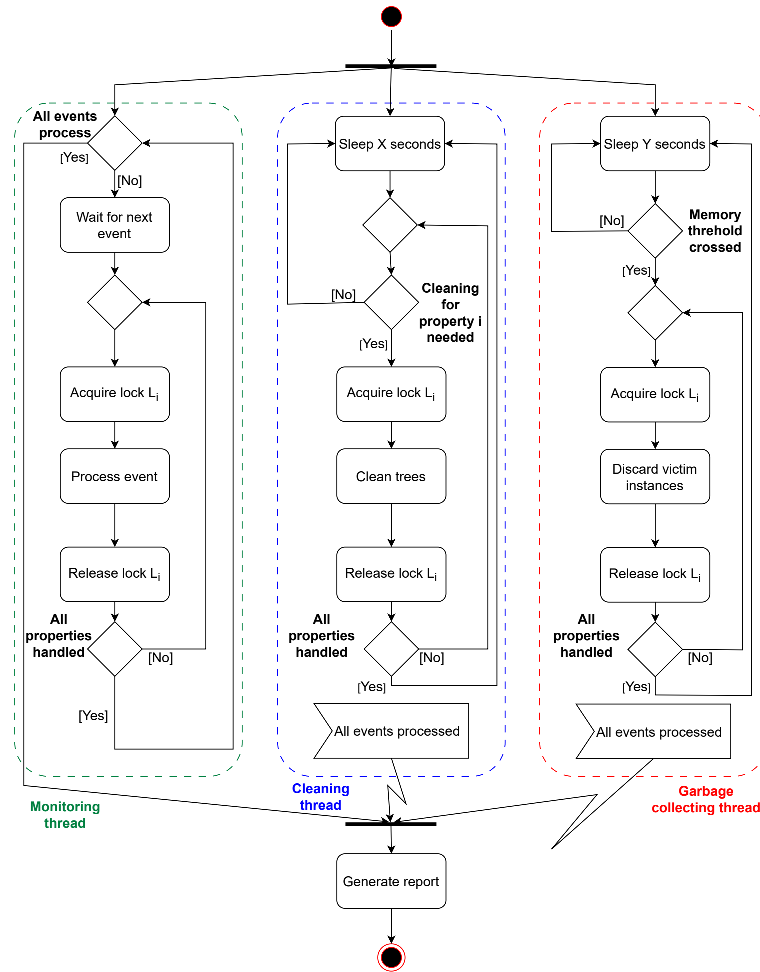


Figure 5.8: Activity diagram covering collaboration between individual threads in *Plogchecker 3.0*.

is removed from the pool. If the lock is not obtained within the timeout, the property is temporarily skipped, and the thread moves on to the next property in the pool. This process repeats until the pool is empty. Once all events are processed, the thread concludes its work and generates the final report.

Cleaning Thread

The main responsibility of this thread is cleaning unnecessary branches in indexing trees. It gets activated only once per `X` seconds. The cleaning thread gets notification about which parameter values are no longer associated with any monitor instances. This helps to determine which branches in which indexing trees need to be removed. The processing flow of this thread is then straightforward. After it wakes up, it checks the received notifications and determines which properties require cleaning. Then it iterates over those properties and removes no longer needed branches from its indexing trees. After cleaning thread gets notified that all event were processed, it concludes it work.

Garbage collecting thread

In compliance with Requirement 1.4, the user can specify the maximum memory usage limit for the tool. This ensures that resources are allocated within bounds, preventing crashes due to out-of-memory errors. However, enforcing this limit may require discarding unfinished monitor instances, potentially leading to the loss of some violation information (e.g., a monitor instance that would eventually detect a violation might be prematurely discarded). Garbage collection is triggered only when the current memory usage approaches the specified limit. The exact threshold percentage is a user-defined value. This thread activates periodically, once every `Y` seconds. Upon activation, it fetches an approximate snapshot of the memory usage for each property. Since this operation is read-only and does not require the most up-to-date data, it is performed without locking. The thread then iterates over properties whose memory usage exceeds the average among all properties. For each such property, it discards a number of monitor instances based on how close the current memory usage is to the maximum limit. The specific instances to discard are selected using the victim selection strategy specified by the user at tool launch (Requirement 1.5). Like the cleaning thread, the garbage collecting thread terminates its activity once all events have been processed.

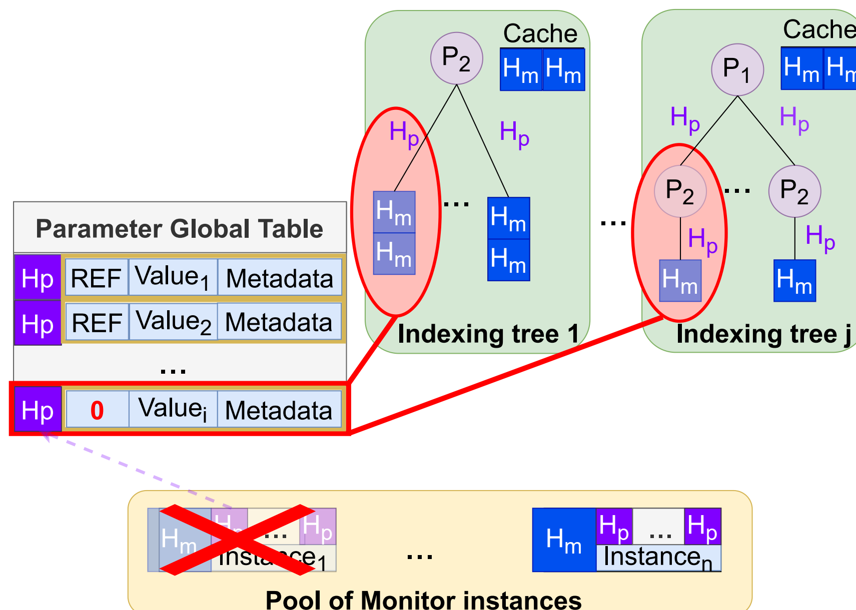


Figure 5.9: Mechanism for the cleaning of the indexing trees.

Cleaning of the Indexing Trees

As described in Section 5.4, the parameter global table contains an item for each parameter value. Each item maintains a counter that tracks the number of monitor instances associated with its parameter value. This counter is incremented whenever a new monitor instance linked to the parameter value is created and decremented when the corresponding monitor instance is discarded. These operations are managed exclusively by the *Monitoring thread*. When the counter for an item reaches zero, it indicates that the parameter

value is no longer associated with any monitor instances, allowing the item to be removed from the global table. Consequently, branches in the indexing trees associated with this parameter value become redundant. However, the *Monitoring thread* does not handle the cleanup of these branches directly to avoid performance overhead. Instead, it sends a notification to the *Cleaning thread*, specifying which parameter values are no longer needed. The *Cleaning thread* then processes these notifications and removes the unnecessary branches from the affected indexing trees. This workflow is illustrated in Figure 5.9.

Preemptive Garbage Collection of Monitor Instance

As previously mentioned, preemptive discarding of monitor instances is sometimes necessary. Requirement 1.5 specifies that multiple strategies for victim selection must be supported. Each strategy requires distinct data structures and internal logic for managing the monitor instance pool. To address this, a hierarchy has been proposed, as illustrated in Figure 5.10. At the top of the hierarchy is a general interface defining the common methods for all pool types. Each strategy is implemented as a separate class, tailored to its specific requirements. The following pool strategies are included in the proposal:

- **Pool with Least Recently Used (LRU) Strategy** maintains monitor instances in a doubly linked list. For efficient lookup, it also includes a mapping of monitor instance hashes to their corresponding elements in the list. When an instance needs to be updated, the pool first locates the linked list element, then moves it to the front of the list. Victims are selected from the elements at the end of the list.
- **Pool with Least Frequently Used (LFU) Strategy** maintains a mapping of monitor instance hashes to wrapper elements containing the instances. Each wrapper also includes a usage frequency counter. Additionally, a separate mapping associates frequency values with linked lists of instances that share the same frequency. Victim selection involves choosing instances from the group with the lowest frequencies.
- **Pool with Random Selection Strategy** simply maintains a mapping of monitor instance hashes to the instances themselves. Victim selection is performed by randomly selecting the required number of instances. This strategy is straightforward and does not require additional data structures or logic, but it does not account for usage patterns when selecting victims.

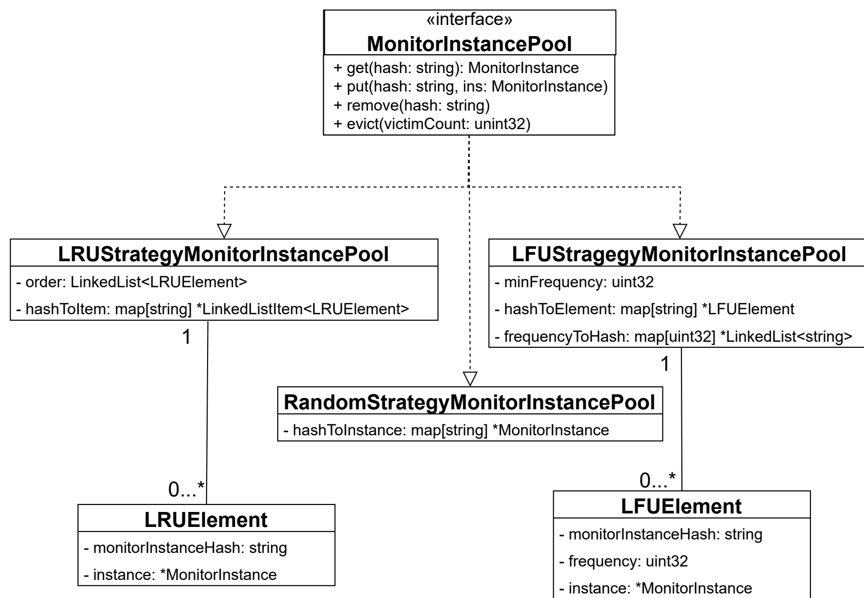


Figure 5.10: Class diagram showing different variants of the strategies for a preemptive garbage collection.

Chapter 6

Implementation Details

This chapter discusses the implementation details of *Plogchecker 3.0*. The main focus is on how the proposed design from Chapter 5 was transformed into the final tool. The initial section (Section 6.1) provides an overview of the general structure, briefly describing the chosen architecture and its division into several independent modules and services. Section 6.2 then describes the module responsible for parsing the configuration file and converting it into the appropriate internal representation. Next, Section 6.3 explains the implementation of the monitoring algorithm. Finally, Section 6.4 reviews each of the services that make up *Plogchecker 3.0*.

6.1 Overview

Plogchecker 3.0 is primarily responsible for reading a stream of input events, checking them for property violations and generating a report of detected breaches. In other words, it functions as a data processing pipeline. To achieve this, the implementation is split into several mostly independent modules. This modular design enables effective concurrency.

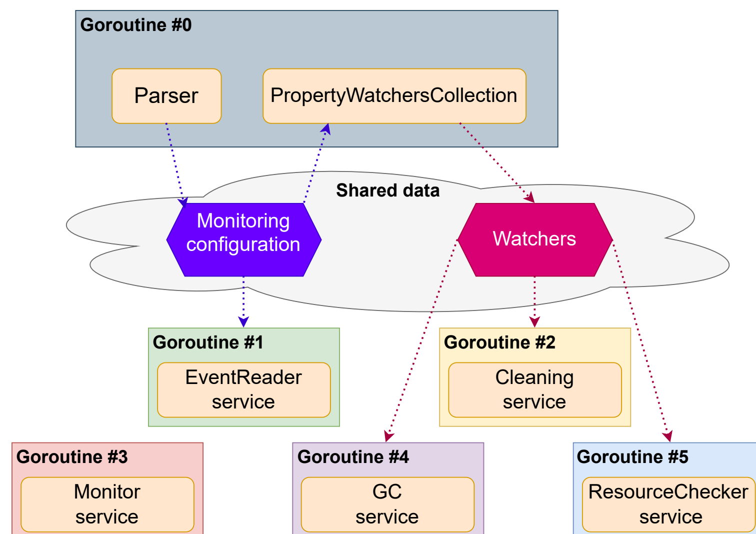


Figure 6.1: Representation of the modules used for the implementation of the *Plogchecker 3.0*.

As discussed in Section 5.2, *Plogchecker 3.0* is implemented in Go, a programming language that provides robust built-in support for concurrent programming. The fundamental concurrency primitives in Go are lightweight threads known as *goroutines*[1]. Several of them are used in *Plogchecker 3.0* to represent the aforementioned modules. A high-level overview of the tool's internal architecture is presented in Figure 6.1.

The main goroutine (*Goroutine #0*) is primarily responsible for transforming the configuration into an appropriate internal representation used for monitoring. This includes creating the *monitoring configuration* (i.e., representations of events, constraints and properties), as detailed in Section 6.2. Using this configuration, it also generates *watchers* that monitor the specified properties, as further explained in Section 6.3. Once both of these shared data structures are prepared, the main goroutine initiates a set of services that collaboratively handle the monitoring process and generate the violation report. Currently, these consist of five services, each running in its own goroutine. Section 6.4 provides a more detailed explanation of these services.

6.2 Configuration Parser

Plogchecker 3.0 uses the same input configuration format as described in Section 5.3. The configuration is provided as a YAML file with sections for defining events, properties and constraints. The processing of this configuration file is handled by the `Parser` type (defined in `parser.go`) which leverages the well-established `go-yaml`[2] library. Internally, the Parser delegates the processing of each section to specialized handlers, as designed in Section 5.3. The result is an internal representation of the configuration suitable for subsequent monitoring tasks.

Events

The *events* section is processed by the `EventHandler` (defined in `event_handler.go`). Raw string event definitions from the configuration are transformed into an internal representation as described in Section 5.3. For each event, an `EventTemplate` is generated, containing its identifier and details about its parameters (Requirements 2.4 and 2.5).

Several validation rules are enforced during EventTemplate generation. Event identifiers must match the following regular expression: `^\w+$`. Additionally, at least one event must be defined. If these conditions are violated or if there is a syntax error (e.g., an unknown parameter data type), the configuration is deemed invalid and *Plogchecker 3.0* terminates.

Each event parameter must have a data type (Requirement 2.7). The full list of supported types is provided in Table 6.1. In *Plogchecker 2.0*, some types were specific to constraints (e.g., `BOOL` or `DURATION`). In *Plogchecker 3.0*, all supported types are now available for event parameters.

[1] Lightweight threads managed by the Go runtime—https://go.dev/tour/concurrency/1.
[2] Package for management of YAML values in Go programs—https://github.com/go-yaml/yaml.

<table>
<tr><th>Keyword</th><th>Values regex specification</th><th>Description</th></tr>
<tr><td>%{NUMBER}</td><td>'0|(-?[1-9]\d *)'</td><td>Positive and negative integers.</td></tr>
<tr><td>%{WORD}</td><td>'"([!#- \t]|\")*"'</td><td>Quoted string values (allowing basic escape and whitespace character usage).</td></tr>
<tr><td>%{BOOL}</td><td>'(true)|(false)'</td><td>Boolean values.</td></tr>
<tr><td>%{DATE}</td><td>ISO8601 or RFC1123 formats</td><td>Dates represented in ISO8601 or RFC1123 formats.</td></tr>
<tr><td>%{DURATION}</td><td>'(\d)+h:(\d)+m:(\d)+s'</td><td>Elapsed time between two dates.</td></tr>
<tr><td>%{IP}</td><td>IPV4 or IPV6 formats</td><td>IPV4 and IPV6 addresses.</td></tr>
<tr><td>%{PATH}</td><td>'((/[\w_%!$@:.+ -]*)+)|
((([A-Za-z]+:)|\)(\[^?*,]*)+)'</td><td>UNIX and Windows absolute paths.</td></tr>
</table>

Table 6.1: Data types which can be used for the parameter specification in *Plogchecker 3.0*.

The list of supported data types has been enhanced. The string type (i.e., `WORD`) has been generalized from single words to quoted strings that allow whitespace and basic escape sequences. Dates are now represented by a single `DATE` type supporting multiple date formats. Two entirely new data types have been introduced:

- `IP`: for representing IPv4 and IPv6 addresses.
- `PATH`: for representing absolute filesystem paths (both UNIX and Windows).

Properties

The management of good and bad *properties* (as requested by Requirement 2.6) is the responsibility of the `PropertyHandler` (defined in `property_handler.go`). Each property in the configuration file is converted into an internal representation, as described in Section 5.3. Similar to events, property identifiers must match the regular expression `^\w+$` and must be unique across all properties (both good and bad). At least one property (good or bad) must be defined. If any of these rules are violated, or if a property contains a syntax error (e.g., an invalid operator or a reference to an undefined event), the parsing process terminates with an error.

The grammar for parsing a single property is shown in Listing 6.1. It defines a language for expressing regular expressions with several slight modifications. As in classical regular expressions, the grammar includes three fundamental operators: *iteration*, *concatenation*, and *alternation*, with the standard precedence order (*iteration* > *concatenation* > *alternation*).

```
AlternationExpr = ConcatenationExpr { "|" ConcatenationExpr } ;
ConcatenationExpr = IterationExpr { IterationExpr } ;
IterationExpr = Basic IterationOp ;
Basic = "(" AlternationExpr ")" | event ;
IterationOp = "*" | "+" | "?" | "!" | "{" positiveInt "}"
               | "{" "," positiveInt "}"
               |"{" positiveInt "," positiveInt "}" | "" ;
```

Listing 6.1: Grammar used for parsing property definition, expressed in the extended Backus-Naur Form.

One modification is the support for additional iteration operators. Along with the standard Kleene star (`*`), it supports other well-known operators that are used, for example, in the POSIX standard (as mentioned in Section 2.3). That including positive iteration (`+`), bounded iterations (`{m}`, `{m,n}` and `{,n}`), and the optional operator (`?`). Additionally, the grammar introduces a cut operator (`!`) as specified in Requirement 2.12. It is treated similarly to other iteration operators at the grammar level.

After a property's raw string representation is parsed into an AST, Algorithm 2.4.1 is applied. However, this algorithm operates only on basic RE components—concatenation, alternation, iteration, ε, or a single symbol. Therefore, preprocessing is performed to transform AST nodes representing non-standard RE operators into their equivalent forms using only the default RE components. This step mainly applies to the additional iteration operators. Let R be a regular expression as described in Definition 2.3.1, the following rewriting rules are used for the transformation:

- $R^+ \to R \cdot R^*$,
- $R? \to R + \varepsilon$,
- $R\{n\} \to \underbrace{R \cdot \ldots \cdot R}_{n \text{ times}}$,
- $R\{m,n\} \to \underbrace{R \cdot \ldots \cdot R}_{m \text{ times}} + \underbrace{R \cdot \ldots \cdot R}_{m+1 \text{ times}} + \ldots + \underbrace{R \cdot \ldots \cdot R}_{n \text{ times}}$,
- $R\{,n\} \to \varepsilon + R + R \cdot R + \ldots + \underbrace{R \cdot \ldots \cdot R}_{n \text{ times}}$,
- $R! \to \varepsilon + R$.

The NFA and DFA representations are illustrated in the class diagram in Figure 6.2. Both automata consist of a pointer to the start state, a mapping of state identifiers to state representations and an internal counter for generating unique state identifiers. The state structures are similar for both automata. The main difference is that `NfaState` supports ε-transitions and multiple transitions per symbol, whereas `DfaState` allows only a single transition per symbol. Both state types share a state identifier and two helper flags. The first one indicates whether the state is final. The second flag serves for marking the resulting states of the expressions associated with the cut operator (i.e., the *cut states*).

A *cut state* is similar to a final state but represents a failed match—when reached, the automaton halts and the expression is considered unmatched. During determinization, *cut states* are handled similarly to final states. If a DFA state represents a set of NFA states containing at least one *cut state*, the DFA state is also marked as *cut*.

The existence of cut states must also be accounted for when minimizing the DFA. Therefore, the minimization algorithm (Algorithm 2.4.3) is slightly modified. Instead of starting with two initial partitions (final and non-final states), the adjusted algorithm uses three partitions—final states (including *cut states* that are also final), *cut states* (non-final) and all remaining states. The rest of the minimization process remains unchanged.

Constraints

The relationships between parameters are expressed through constraints (Requirement 2.9). All operations involving constraints are managed by the `ConstraintHandler` (defined

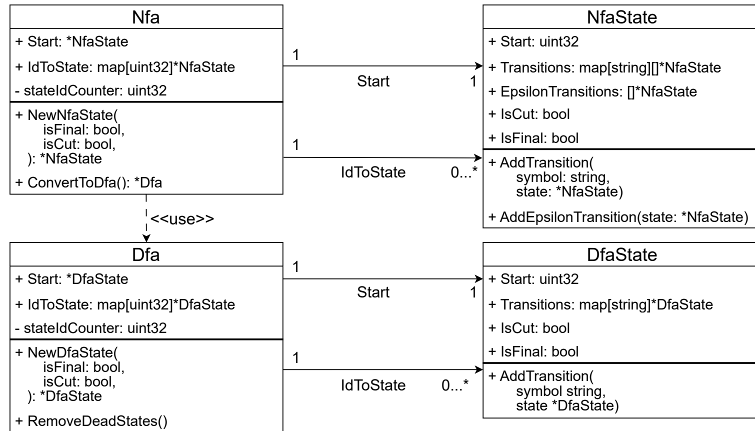


Figure 6.2: Class diagram demonstrating the structure of the NFA and DFA for representing the properties.

in `constraint_handler.go`). It parses the *constraints* from the configuration file into an internal representation and performs optimizations as described in Section 5.3. If a constraint cannot be parsed (e.g., due to a syntax error or a reference to a non-existent parameter), it is simply skipped and excluded from the final set of constraints. In such cases, the tool's execution is not terminated.

The conversion of a constraint's string representation into AST nodes is performed using the grammar shown in Listing 6.2. Constraints are represented as expressions that can include various arithmetic operators (e.g., `+`) and relational operators (e.g., `>=`) as specified in Requirement 2.9. They can also include a set of built-in functions (Requirement 2.10) which are discussed in detail below. Additionally, literals of all supported data types and parameters from events can serve as operands in these expressions (Requirement 2.11).

```
Start = FirstLevelExpr ;
FirstLevelExpr = SecondLevelExpr { SecondLevelExpr } ;
FirstLevelOp = ">" | "<" | ">=" | "<=" | "=" | "!=" ;
SecondLevelExpr = Term { SecondLevelOp Term } ;
SecondLevelOp = "+" | "-" ;
Term = bool | date | ip | path | int | string | duration | parameter
     | "(" FirstLevelExpr ")" | funcIdentifier "(" FuncArgs ")" ;
FuncArgs = [ FirstLevelExpr { "," FirstLevelExpr } ] ;
```

Listing 6.2: Definition of the grammar for parsing constraint definitions, written in Extended Backus-Naur Form.

Table 6.2 lists the supported built-in functions. This set includes string functions already available in *Plogchecker 2.0*, as well as two newly added functions for operations on the `IP` and `PATH` data types.

Once the raw constraints are converted into their AST representation, several semantic checks are performed. The primary goal is to ensure that all operators and functions are applied to operands of appropriate types. Furthermore, the evaluated result of any constraint expression must be of type `BOOL`. Type evaluation of AST nodes is done recursively. Prameters and literals have inherently known types, function parameter and return

Function signature	Description
`is_substr(base:WORD, str:WORD):BOOL`	Checks if `str` is a substring of `base`.
`length(str:WORD):NUMBER`	Gets the number of characters in `str`.
`is_parent_dir(parent:PATH, child:PATH):BOOL`	Checks if `parent` is a parent path to `child`.
`prefix(addr:IP, length:NUMBER):IP`	Gets subnet IP from the given **addr** for the provided prefix `length`.

Table 6.2: Built-in functions which can be used in the constraints.

types are defined in their signatures (see Table 6.1) and operators are validated according to the rules outlined in Appendix B.

6.3 Property Watchers

The property watchers provide an abstraction layer for executing the monitoring process. They are constructed using the property specifications parsed by the `Parser` (Section 6.2). A separate watcher is created for each defined property. Conceptually, a property watcher represents a parametric property (Definition 2.2.7). Each watcher contains a template monitor that defines the property pattern. To correctly handle *trace slices* across different parameter bindings, the watcher manages multiple instances of this template monitor, with each instance dedicated to a specific binding. These instances are created and updated based on the events processed by the watcher. Event processing within a watcher is carried out using the monitoring algorithm described in Section 5.4.

Property Watchers Collection

Plogchecker 3.0 is designed to support the simultaneous monitoring of multiple properties (Requirement 1.1), each with an optional number of parameters (Requirement 1.2). Since each property requires its own watcher, the system may need to manage several watchers at once. To facilitate this, the `PropertyWatchersCollection` type (defined in `property_watchers_collection.go`) abstracts the management and operations over multiple property watchers.

Figure 6.3 shows the class diagram illustrating the `PropertyWatchersCollection` type and its managed items. Its responsibilities are straightforward—it provides iterators over the collection of watchers. The first iterator allows iteration over all watchers, while the second iterator returns only the watchers that are relevant to a given event—i.e., watchers that manage properties whose patterns include the event.

As briefly mentioned in Section 6.1, property watchers are shared resources used by multiple services that contribute to the overall processing in *Plogchecker 3.0*. Since these services run concurrently, they compete for access to the property watchers. To ensure correctness, appropriate synchronization mechanisms are required. For this reason, the `PropertyWatchersCollection` does not directly manage property watchers but instead manages `PropertyWatcherItem` objects. These objects act as wrappers around the property watchers and provide mechanisms for enforcing exclusive access.

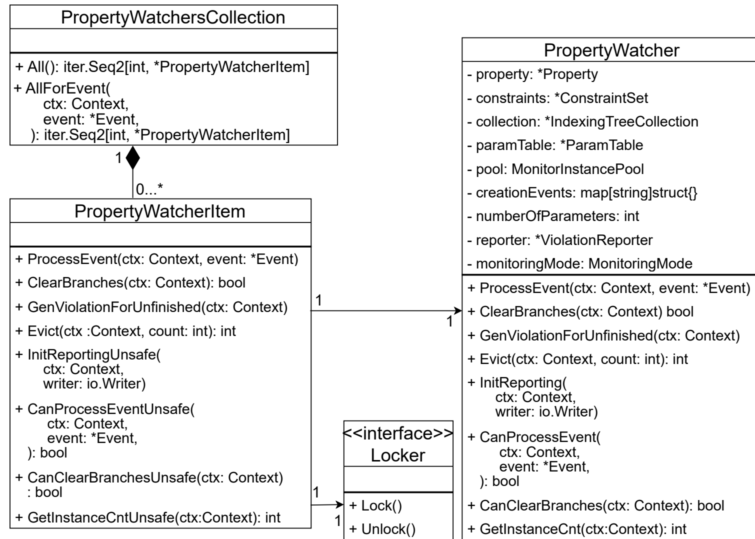


Figure 6.3: Class diagram representing the structure of the property watchers collection.

Exclusive access in `PropertyWatcherItem` is achieved through an attribute implementing the `Locker` interface, which provides classical mutex lock behavior. Another attribute of `PropertyWatcherItem` is the `PropertyWatcher` itself. When an operation needs to be performed on a watcher, the `PropertyWatcherItem` first acquires the associated mutex. Once the operation is complete, the mutex is released.

However, some operations on property watchers do not require exclusive access. Examples include one-time configuration tasks (e.g., initialization of violation reporting) or read-only actions where up-to-date data is not strictly required (e.g., retrieving the current number of monitor instances). To distinguish these operations from those requiring locking, they are marked with the `Unsafe` suffix.

The final part of the class diagram is the `PropertyWatcher` type which represents the watcher itself. It includes pointers to the monitor instance *pool*, *parameter table* and *indexing trees* as described in Section 5.4. Additionally, it contains the pointer to the concrete implementation of the *monitoring mode* (Section 5.5), which can be selected via the `--monitoring-mode` parameter (defaulting to *Standard* mode). Finally, it includes a *reporter* responsible for generating and processing detected violations.

Garbage Collection Strategies

In accordance with Requirement 1.5, property watchers ensure that resource utilization remains within reasonable bounds. This is achieved by selectively discarding (i.e., preemptively garbage collecting) a number of in-progress monitor instances on demand.

The decision of when to select victims and how many to discard is made outside of the property watchers. This responsibility lies with one of the services (details are provided in Section 6.4). The property watchers only determine which monitor instances are chosen as victims. The primary goal is to minimize information loss by selecting the suitable victims, as discarded monitor instances might otherwise have detected violations. To achieve this, specialized victim-selection strategies are implemented within the monitor instance pools (as proposed in Section 5.4).

To support this, a common interface is defined for the monitor instance pool, with different implementations providing various victim-selection mechanisms. *Plogchecker 3.0* supports three strategies:

- *Random*: Victims are chosen using uniform random sampling.
- *LRU*: Victims are chosen based on the least recently used criterion.
- *LFU*: Victims are chosen based on the least frequently used criterion.

The concrete pool implementation is selected at the time of property watcher creation and is shared across all watchers. This selection is based on the value provided via `--garbage-collection-strategy` parameter when launching *Plogchecker 3.0*.

Violation Reporting

The primary responsibility of the property watchers is to report any detected property violations. *Plogchecker 3.0* is designed to provide information about violations as soon as they are encountered. This is achieved by appending a new record to the violation report file, which is shared across all property watchers. The file uses the Newline Delimited JSON (NDJSON)[3] format which is particularly well-suited for scenarios where items are continuously appended—such as data streams or log files. This makes it an ideal choice for recording violations, as each violation report is represented by a single JSON object on a single line, terminated by a newline character.

The JSON schema of a single violation report is illustrated in Listing 6.3. It contains basic information about the property to which the violation belongs (e.g., its identifier and a flag indicating whether the property is good or bad). Additionally, it includes the representation of the *trace slice*, consisting of the events that led to the violation. Each event is described by its identifier and all associated parameters.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Violation report",
  "description": "Detailed report of~the~property violation",
  "type": "object",
  "properties": {
    "is_good_property": { "type": "boolean" },
    "property_id": { "type": "string" },
    "trace": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "event_id": { "type": "string" },
          "parameters": {
            "type": "array" },
            "items": {
              "type": "object",
              "properties": {
```

[3]Specification of the NDJSON format—https://github.com/ndjson/ndjson-spec

```
                "param_id": { "type": "string" },
                "raw_value": { "type": "string" },
                "type": { "type": "string" },
              },
              "required": ["param_id", "raw_value", "type"]
            }
          },
        "required": ["event_id", "parameters"]
      }
    }
  },
  "required": ["is_good_property", "property_id", "trace"]
}
```

Listing 6.3: JSON schema definition of the single violation report.

The procedure for generating a violation report depends on whether the property watcher monitors a good or a bad property. For bad properties, the process is straightforward. A report for a given *trace slice* is generated as soon as the corresponding monitor instance transitions into its final state. For good properties, however, reports are generated whenever a monitor instance fails to reach its final state. This situation can arise when a monitor instance is prematurely terminated due to the application of the cut operator. In this case, a violation report can be generated immediately. Alternatively, reports for unfinished monitor instances are generated when the monitoring process terminates (i.e., when all events have been processed).

6.4 Services

The main monitoring process in *Plogchecker 3.0* is managed by several concurrently running services. All services implement a common interface to ensure consistent and idempotent usage. This interface consists of a single `Start()` function, responsible for initializing and launching the service.

Once the monitoring configuration and property watchers are ready, the main goroutine starts all services and then blocks until they complete. This synchronization is implemented using the `WaitGroup` construct from Go's `sync`[4] package which waits for a group of goroutines to finish.

To coordinate the services, *Plogchecker 3.0* uses a shared `Context` from the `context`[5] package. It is primarily used to signal when services should terminate due to an external notification (e.g., `SIGINT` signal) or when all input data has been processed.

Event Reader Service

The event reader is one of the most critical services. It reads and parses raw input events, converting them into structured representations based on predefined event templates and forwards them to other services. The service reads from standard input (*stdin*), supporting both offline monitoring (Requirement 2.3) via file redirection and online monitoring (Requirement 2.2) when the monitored system produces events in real time.

[4] Package providing basic synchronization primitives—https://pkg.go.dev/sync.
[5] Package providing means for sharing information across API boundaries—https://pkg.go.dev/context.

Figure 6.4 shows the class diagram of the service. The `EventReader` parses raw events by matching them against templates provided by `EventParameterMap`. Parsed events are then sent to other services via a Go channel[6], a standard mechanism for inter-goroutine communication.

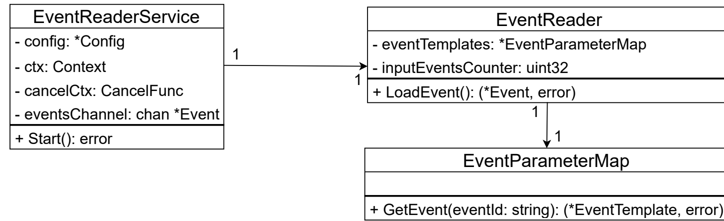


Figure 6.4: Class diagram representing the event reader service.

Once all input events have been processed (i.e., when `EOF` is reached), the service performs cleanup. It closes the events channel to notify consumers that no more data will arrive. Additionally, the shared context is used to signal other services that they can proceed with termination since all input data has been processed.

The processing flow of the event reader service is illustrated in Figure 6.5. It reads raw event representations formatted as tuples, where individual parts are separated by whitespace. The first item always represents the event identifier, while the remaining items represent the parameter values associated with the event. These parameter values are expressed as literals of the supported data types, as defined in Table 6.1. If a raw event cannot be matched with any of the event templates (e.g., due to an incorrect identifier or mismatched number or types of parameters), it is simply ignored and processing continues with the next event. If the `EventReader` successfully parses an event, the internal event counter is incremented and the parsed event is sent through the events channel for further processing by other services.

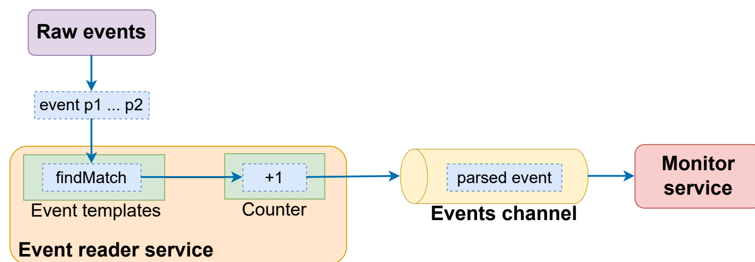


Figure 6.5: Data processing flow in the event reader service.

Monitor Service

Another very important service is the monitor service. It is responsible for managing property satisfaction checks by applying the input events to the property watchers. Its representation in the form of a class diagram can be found in Figure 6.6.

This service accepts the property watchers collection (Section 6.3) represented by the `WatchersCollection` interface. First, it configures all property watchers to report

[6]Typed conduit for sending and receiving data—https://go.dev/tour/concurrency/2.

Figure 6.6: Class diagram representing the monitor service.

violations in the shared report file. Then, it processes events from the events channel by forwarding them to the relevant watchers. After all events are processed, the service iterates over all watchers and requests violation reports for any unfinished monitor instances, if applicable.

It is important to note that the monitor service is not terminated immediately based on information from the shared context. Instead, it stops only after all data from the events channel is processed (i.e., the channel is closed by the producer). This ensures that even if services are terminated due to an external notification, the monitor service processes all events read up to that point.

The actions during event processing are visualized in Figure 6.7. The service first takes an event from the events channel, then iterates over all property watchers relevant to this event—i.e., watchers whose property patterns include it. These watchers then generate violation reports in the resulting file, if needed.

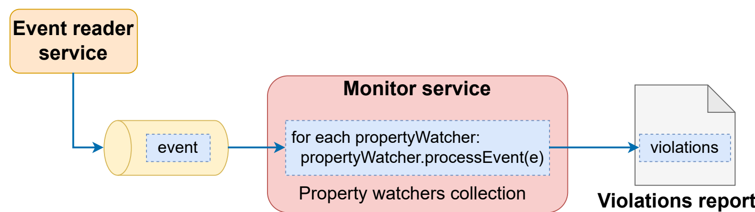


Figure 6.7: Data processing flow in the monitor service.

Cleaning Service

The main responsibility of the cleaning service is to manage resource release during monitoring. Its overall structure is shown in the class diagram in Figure 6.8.

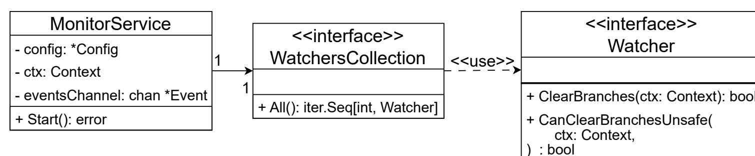


Figure 6.8: Class diagram representing the cleaning service.

This service uses the property watchers collection (Section 6.3), represented by the `WatchersCollection` interface. It iterates over all property watchers and checks for redundant branches in their indexing trees, which can then be removed (following

the logic described in Section 5.4). These checks are performed without locking the watchers for performance reasons since the cleaning operation is non-critical and tolerates slightly outdated values. If redundant branches are found, the watcher is locked and cleaned, requiring exclusive access.

Cleaning is performed periodically, with the interval controlled by the `--cleaning-interval` parameter, specified in seconds (default: 20 s). After completing one cleaning round, the service sleeps until the next cycle.

Garbage Collector Service

The garbage collector (GC) service periodically monitors *Plogchecker 3.0*'s memory usage and preemptively discards monitor instances when usage exceeds configured limits. This service is not started automatically. It must be explicitly enabled by selecting a garbage collection strategy via the `--garbage-collection-strategy` parameter. If the value is anything other than *None* (default), the GC service is started.

Its behavior is further configured by several command-line parameters:

- `--garbage-collection-interval`: defines how often checks occur (default: every 20 s).
- `--memory-limit`: specifies the maximum memory usage. This is not a hard limit, as memory usage can temporarily exceed it between GC rounds.
- `--memory-usage-threshold`: defines the fraction of the limit at which preemptive discarding begins. This prevents unnecessary evictions early on since discarding monitor instances can lead to lost future violations. The fraction must fall within the range 0.5–1.0.

The GC service is represented in Figure 6.9. It relies on a `GarbageCollector` instance to perform each GC round. The `GarbageCollector` in turn uses an attribute satisfying `MemoryChecker` interface to obtain the current memory usage. The implementation of this interface leverages the `gopsutil`[7] package, using the Resident Set Size (RSS) as the memory usage metric. The `GarbageCollector` also interacts with the property watchers to evict monitor instances.

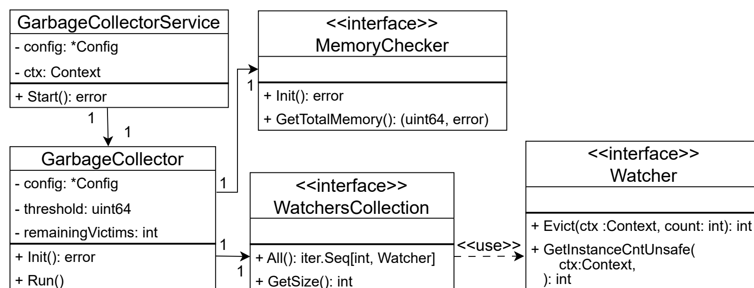


Figure 6.9: Class diagram representing the garbage collector service.

During each GC round, the service:

[7] Package providing port of `psutil` for Go—https://github.com/shirou/gopsutil.

1. Determines current memory usage.

2. If usage exceeds the configured threshold, calculates the portion of monitor instances to evict using the following formula:

$$v = \begin{cases} 0 & \text{if } m_c \leq m_t \\ C_1 \cdot \left(\frac{m_c - m_t}{m_m - m_t}\right)^3 & \text{if } m_t < m_c < m_m \\ 1 - \frac{C_2 \cdot m_m}{m_c} & \text{otherwise} \end{cases}$$

   where m_c denotes the current memory usage, m_t the configured threshold for GC activation, m_m the configured memory limit, C_1 the maximum portion of victims (empirically set to 0.1) and C_2 the target rebound ratio for memory usage after exceeding the memory limit (empirically set to 0.9).

3. Retrieves the total number of monitor instances (N_i) across watchers. The victims count ($v \cdot N_i$) is divided proportionally among watchers based on their respective number of instances.

4. Iterates over all watchers, instructing each to evict its allocated share of monitor instances.

Resource Checker Service

The resource checker service is primarily diagnostic and not required for the tool's core functionality. Its responsibility is to periodically collect resource usage metrics during execution and save the results to a CSV file. Currently, it tracks only one metric—total memory usage.

An overview of this service is shown in Figure 6.10. Like the GC service, it uses attribute implementing the `MemoryChecker` interface to retrieve memory usage, which is then written to the reporting file along with a timestamp.

Figure 6.10: Class diagram representing the resource checker service.

This service is disabled by default. It is launched only when a report file name is provided via the `--resource-checks-file` parameter. Once active, it collects and logs resource utilization every 3 s.

Chapter 7

Testing and Evaluation

This chapter presents the experimental evaluation of the *Plogchecker 3.0* implementation, as introduced in Chapter 6. The primary focus is on a careful assessment of both its functionality and performance. Section 7.1 evaluates the performance of the tool with respect to various scaling factors. Section 7.2 provides a detailed comparison of the implemented garbage collection strategies, with particular emphasis on their effectiveness in reducing memory usage while minimizing the number of missed violations caused by premature monitor instance eviction. Finally, Section 7.3 addresses the functional verification, ensuring that the tool satisfies all specified requirements.

7.1 Scaling Benchmarks

This section evaluates the performance of *Plogchecker 3.0* using a set of scaling benchmark tests. The primary goal was to assess how various aspects of monitored properties influence the tool's performance (Requirements 1.1 and 1.2). Garbage collection was intentionally disabled during the benchmarks to measure performance under full load. Preemptive discards of monitor instances, if enabled, could significantly affect the results, as poorly chosen victims might reduce the processing workload and distort the metrics. Therefore, the evaluation of garbage collection is covered separately in Section 7.2.

Overview

To support a systematic testing approach, several parameters were defined to represent key factors contributing to monitoring complexity. The following parameters were used:

- Number of parameters per property,
- structural complexity of the properties,
- number of monitored properties.

Property complexity was divided into three categories to reflect the structural intricacy of the patterns while, as much as possible, keeping the number of events consistent. The categories are (defined by using the *Plogchecker 3.0*'s syntax for properties):

- **Simple**: `a b c`,
- **Moderate**: `a (b | c) d`,

- **Complex**: `a (b | c)+ d{2}`.

The benchmark suite was divided into three test groups. In each group, all but one parameter were fixed to isolate and evaluate the impact of a single scaling factor. Each test group was executed separately for both implemented monitoring modes. This allowed for a direct comparison of how each mode responds to the scaling parameters.

Another important consideration was defining appropriate input sequence sizes to ensure comparability within each test group. Two strategies were used depending on the scenario. The first approach fixed the total number of events in the input sequence. However, this is not always ideal, so a second approach was introduced—fixing the total number of subsequence bundles per property. This method better captures scenarios where multiple properties are used.

Performance was evaluated using two metrics—elapsed wall-clock time and maximum resident set size (RSS). Both values were collected using the `/usr/bin/time`[1] utility. To reduce noise in the measurements, each test was run five times and the final results were computed as the average across these runs. The hardware and environment specifications for the benchmarks are provided in Table 7.1.

Component	Details
CPU	Intel Core i5-8350U @ 1.70 GHz
Cores / Threads	2 cores × 2 threads (4 logical CPUs)
Architecture	x86_64
RAM	10 GiB
Swap	2 GiB
Operating System	Ubuntu 22.04.5 LTS (Jammy)
Environment	WSL2[2]
Kernel	6.6.87.2-microsoft-standard-WSL2
Go Version	go1.24.4 linux/amd64

Table 7.1: Specification of the environment used for benchmarks.

Generation of Data for Tests

The data for the scaling benchmark tests were generated using the helper Python script `benchmark_data_generation.py`. This script accepts a test specification based on the parameters described previously. It produces two types of output:

YAML configuration file for *Plogchecker 3.0*

This file defines the properties and parametric events according to the provided test specification. The total number of parameters is distributed evenly among the events. The constraints section consists exclusively of expressions enforcing parameter equivalence, ensuring that *Plogchecker 3.0* is required to perform monitor instance duplication during the tests.

Event sequence file

This file contains the event trace for the benchmark. Events are generated to include

[1] A simple command for retrieving resource usage—https://linux.die.net/man/1/time.
[2] Windows Subsystem for Linux—https://learn.microsoft.com/en-us/windows/wsl/about.

a variety of parameter values. When possible, related parameters are combined using the Cartesian product. To keep the number of combinations within reasonable bounds, random sampling is applied to the resulting tuples. Furthermore, the structure of each property pattern is respected during generation—all branches of alternation operators are covered, and a range of repetition counts is used for iteration operators.

Event generation is performed separately for each property and proceeds in rounds. In each round, the property's pattern is traversed sequentially and events are generated to cover different execution paths—both in terms of parameter values and pattern structure.

The number of rounds is determined by the specified input size. If the input size is defined as the total number of events, that number is distributed evenly across all properties. For each property, as many rounds as possible are executed until the event limit is reached. Alternatively, if the input size is specified by the number of subsequence bundles, then the number of rounds for each property is set directly to the number of bundles.

Scaling with Number of Parameters

The first test group focused on scaling with respect to the number of parameters per property. The test cases were configured with the following settings:

- **Number of parameters per property**: variable (0–13 range)
- **Structural complexity of the properties**: *Simple*
- **Number of monitored properties**: 1
- **Input size**: 50,000 total events

The results for both monitoring modes supported by *Plogchecker 3.0* are shown in Figure 7.1. The upper plots depict runtime performance. Interestingly, in the lower parameter range, the standard mode was initially slower (upper right plot). However, since the absolute runtime values were extremely low in this region, the differences amounted to fractions of a second and are negligible.

The more significant trend appears from 5 parameters onward—runtime began increasing rapidly and the *No Out of Order Event* (*NOoOE*) mode consistently exhibited worse performance with peaking at about 10% slower. The upper left plot reveals that runtime initially grew linearly with a shallow slope, but transitioned into exponential growth between 8 and 11 parameters.

The lower plots show maximum RSS. As the lower right plot demonstrates, memory usage remained nearly identical across both modes for all parameter counts (differences within 1–2%). Memory consumption began increasing more sharply after 8 parameters and surged notably between 11 and 12 parameters.

Scaling with Property Complexity

The second test group focused on how increasing property complexity affects performance. As before, one parameter varied while the rest remained constant:

- **Number of parameters per property**: 5

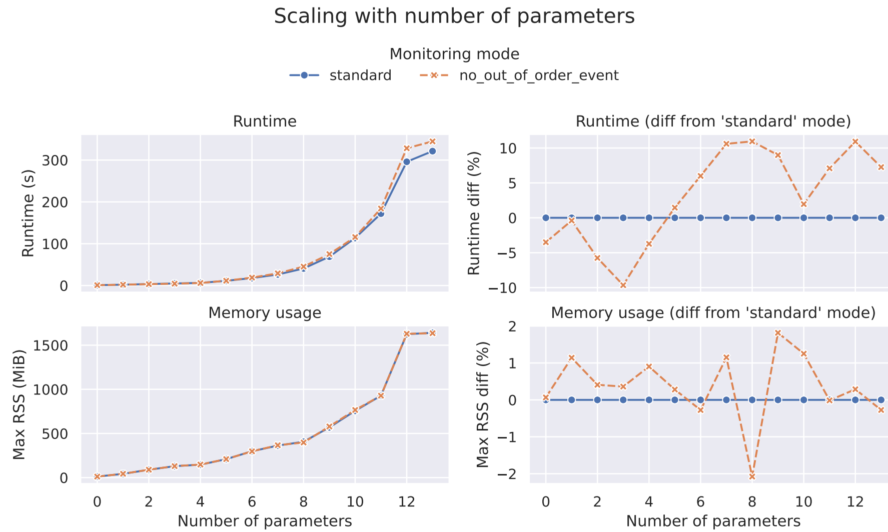


Figure 7.1: Visualization of the runtime and memory usage scaling in regards to the number of parameters.

- **Structural complexity of the properties**: variable—combinations of four property patterns, ranging from all *Simple* to all *Complex* (i.e., `s + s + s + s`, ..., `c + c + c + c`)
- **Number of monitored properties**: 4
- **Input size**: 2,000 subsequence bundles per property

Figure 7.2 shows both runtime and memory usage results. On the x-axis, complexity combinations are labeled using the initial letters of their constituent types (i.e., `s`, `m` and `c`).

As seen in the upper right plot, the *NOoOE* mode was consistently slower, with runtime differences mostly around 5%. The upper left plot shows a nearly linear trend in runtime growth as property complexity increased.

Regarding memory, differences were generally small—again within 2%—except for a notable spike in the `s + s + s + m` configuration (lower right plot). The lower left plot reveals an initially steep linear memory growth that slowed beginning from the combination `m + m + m + c`.

Scaling with Number of Properties

The final test group examined how performance scales with the number of properties. To capture both extremes, it was split into two parts:

- **Part 1**: No event sharing across properties.
- **Part 2**: Properties with overlapping events.

Both parts used the following parameters:

- **Number of parameters per property**: 5

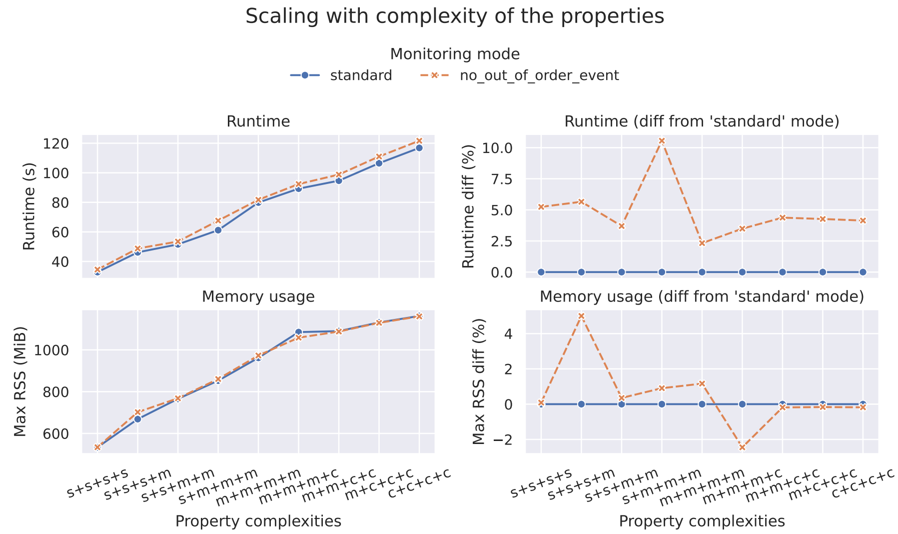


Figure 7.2: Visualization of the runtime and memory usage scaling in regards to the complexity of the properties.

- **Structural complexity of the properties**: *Simple*
- **Number of monitored properties**: variable (range 1–15)
- **Input size**: 2,000 subsequence bundles per property

Figure 7.3 shows the performance results in the non-sharing case. Both runtime, as well as memory usage grew linearly with the number of properties. The *NOoOE* mode was consistently slower, with execution times 5–10% longer than in the *Standard* mode. Memory usage differences were minimal—within 1% across all tests.

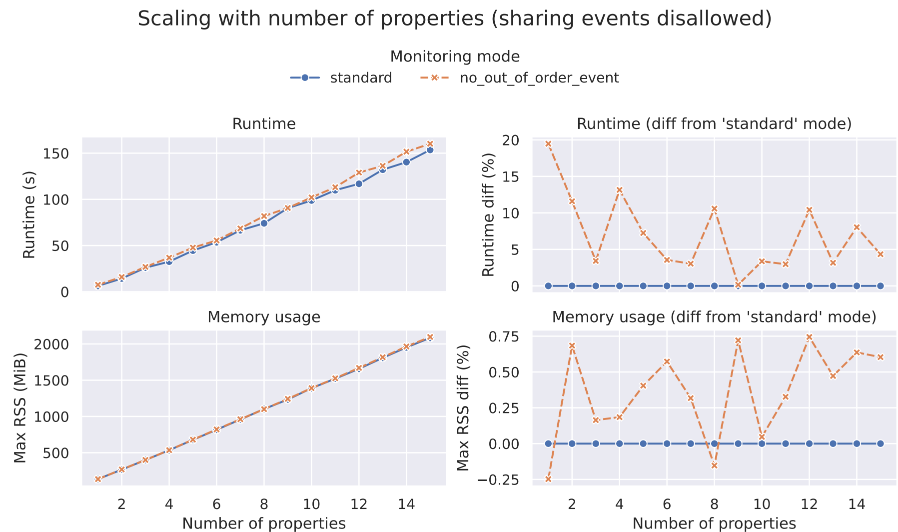


Figure 7.3: Visualization of the runtime and memory usage scaling in regards to the number of properties (without sharing events across properties).

In the event-sharing scenario (results in Figure 7.4), both runtime and memory usage again followed a linear trend. The *NOoOE* mode continued to be slightly slower, with most differences around 5%. Memory usage was again nearly identical across modes, though the *NOoOE* mode used slightly more memory in most cases—by approximately 1%.

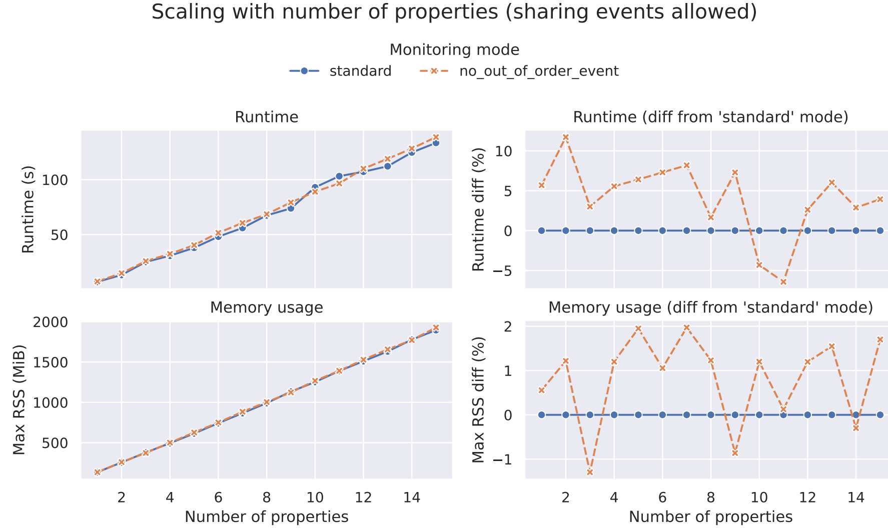


Figure 7.4: Visualization of the runtime and memory usage scaling in regards to the number of properties (with sharing events across properties).

Summary

The scaling benchmarks demonstrated the strong scalability of *Plogchecker 3.0*. In test groups focused on property complexity and quantity (both with and without event sharing), runtime and memory usage exhibited near-linear growth. Similarly, scaling with respect to the number of parameters showed linear trends in both metrics up to eight parameters. However, beyond this point, runtime and memory usage increased rapidly, resembling exponential growth. This behavior is expected, as the growth in parameters eventually leads to state explosion in the managed monitor instances. Nonetheless, this also highlights potential areas for future optimization.

The benchmarks also compared the two implemented monitoring modes. Memory usage was nearly identical across both modes, with differences within 2% across all test groups. In contrast, runtime performance revealed consistent differences—*NOoOE* mode was slower by up to 10%. This overhead is expected due to the stricter runtime constraints imposed by the *NOoOE* mode, which require additional checks and thus increase processing time.

7.2 Comparison of Garbage Collection Strategies

This section evaluates the preemptive discarding of monitor instances in *Plogchecker 3.0*, as required by Requirements 1.4 and 1.5. The primary objective is to provide a systematic comparison of all implemented garbage collection (GC) strategies.

Overview

The input data used for this evaluation was drawn from the scaling benchmarks (Section 7.1). Specifically, the inputs from the final test case in each benchmark group (i.e., representing the most demanding scenarios) were selected. In total, four distinct scenarios were considered:

- A case with a high number of parameters (13 parameters in a single property),
- a case with complex property patterns (4 complex properties),
- a case with a high number of properties without event sharing (15 independent properties),
- a case with a high number of properties sharing events (15 properties with overlapping event sets).

To ensure consistent and fair conditions, a common configuration was applied across all scenarios. The GC interval was fixed at 5 s. The memory usage limit was determined using a uniform strategy. For each scenario, the peak memory usage without GC was first measured. The limit was then calculated as 87.5% of that value.

Each scenario was executed three times to minimize measurement noise, and the results were averaged. All evaluations were also performed separately for both monitoring modes supported by *Plogchecker 3.0*.

The following metrics were recorded for each run:

- Elapsed time,
- maximum resident set size (RSS),
- number of lost violations (i.e., violations that were detected without GC but missed due to preemptive discarding).

An optimal GC strategy is expected to reduce memory consumption while minimizing the loss of violation reports, thus maintaining both efficiency and accuracy.

Evaluation for Scenario with a High Number of Parameters

Standard Monitoring Mode

The baseline measurements without GC and the corresponding memory limit were as follows:

- **Base metrics without GC**: *runtime*=338.66 s, *maximum RSS*=1632.1 MiB
- **Memory usage limit**: 1428 MiB

As in other scenarios, three GC thresholds (0.5, 0.7 and 0.9) were tested. Full results for all the scenarios are provided in Appendix C. Table 7.2 shows the best outcome per strategy for this scenario. Overall, performance improved as the threshold increased, with the 0.9 setting consistently yielding the fewest lost violations. Among the strategies, *LRU* achieved the best balance—maximizing memory utilization while minimizing data loss.

Strategy	Threshold	Runtime (s)	Max memory usage (MiB)	Violations diff (%)
Random	0.9	304.56	1428.36	10.48
LRU	0.9	315.1	1428.09	3.88
LFU	0.9	285.96	1411.66	12.93

Table 7.2: Comparison of GC strategies when applied to the parameter scaling TC (*Standard* monitoring mode).

Figure 7.5 displays memory usage trends for all strategies and thresholds. For each threshold, two reference lines are included—one for the memory usage limit the other for the GC activation threshold. At lower thresholds, memory was kept well below the limit. But it was at the cost of high violation loss. *LRU* effectively maintained memory usage near the limit, while *LFU* often resulted in the lowest runtime and memory usage, albeit with significantly more data loss.

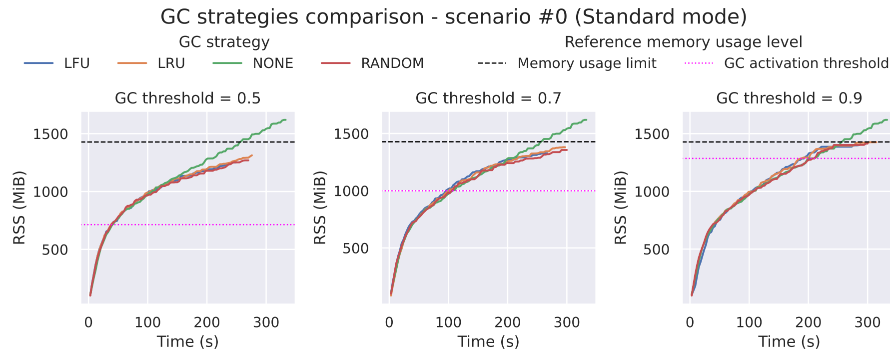


Figure 7.5: Visualization of the memory usages when different GC strategies are applied to the TC with high number of parameters (*Standard* monitoring mode).

No Out of Order Event Monitoring Mode

The baseline measurements without GC and the corresponding memory limit were as follows:

- **Base metrics without GC**: *runtime*=363.08 s, *maximum RSS*=1632.22 MiB
- **Memory usage limit**: 1428 MiB

Best results are summarized in Table 7.3. As in the standard mode, the *LRU* strategy again performed best. Memory usage trends in Figure 7.6 resemble those from standard monitoring.

At the 0.5 threshold, strategies diverged early. *LRU* consumed the least memory for most of the run, though it ended with the longest runtime and highest overall memory use. For 0.7 and 0.9 thresholds, trends aligned closely across strategies, with *LRU* again utilizing memory most effectively.

Strategy	Threshold	Runtime (s)	Max memory usage (MiB)	Violations diff (%)
Random	0.9	331.19	1416.97	10.4
LRU	0.9	341.63	1425.19	4.04
LFU	0.9	311.67	1402.64	12.99

Table 7.3: Comparison of GC strategies when applied to the parameter scaling TC (*NOoOE* monitoring mode).

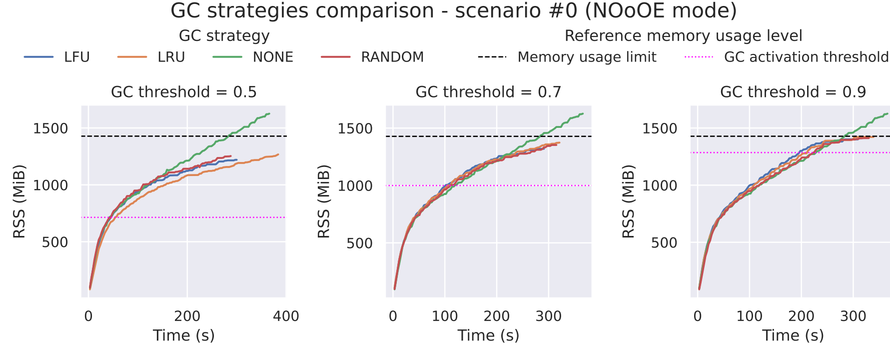


Figure 7.6: Visualization of the memory usages when different GC strategies are applied to the TC with high number of parameters (*NOoOE* monitoring mode).

Evaluation for Scenario with Complex Property Patterns

Standard Monitoring Mode

The baseline measurements without GC and the corresponding memory limit were as follows:

- **Base metrics without GC**: *runtime*=114.37 s, *memory usage*=1163.07 MiB
- **Max memory usage limit**: 1018

Table 7.4 presents the best results. *Random* and *LFU* performed best with threshold 0.9, whereas *LRU* peaked at 0.7. All strategies except *LRU* exceeded the memory usage limit and violation loss was significant—up to 50% for *LFU*, compared to 30% for *LRU*.

Strategy	Threshold	Runtime (s)	Max memory usage (MiB)	Violations diff (%)
Random	0.9	93.59	1064.95	33.99
LRU	0.7	94.28	1004.97	29.64
LFU	0.9	96.82	1114.54	46.27

Table 7.4: Comparison of GC strategies when applied to the property complexity scaling TC (*Standard* monitoring mode).

Figure 7.7 visualizes memory usage. Early GC did not yield lower memory use. *LRU* was the only strategy to stay within the configured limit. Notably, periodic memory drops aligned with the internal cleaning service (Section 6.4), visible even in the baseline.

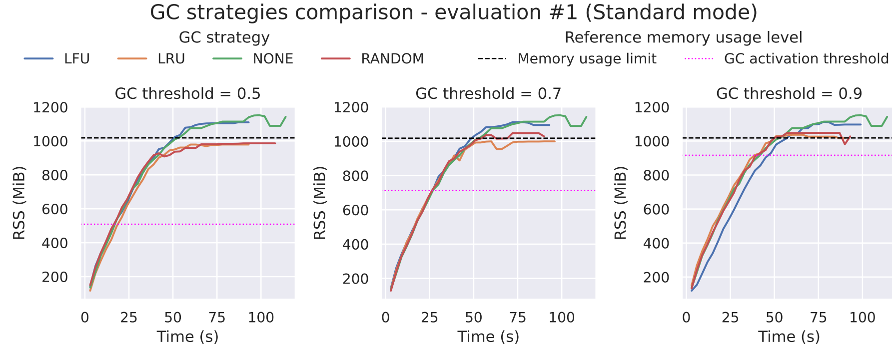


Figure 7.7: Visualization of the memory usages when different GC strategies are applied to the TC with complex properties (*Standard* monitoring mode).

No Out of Order Event Monitoring Mode

The baseline measurements without GC and the corresponding memory limit were as follows:

- **Base metrics without GC**: *runtime*=122.13 s, *memory usage*=1170.31 MiB
- **Max memory usage limit**: 1024

The best results for each strategy can be seen in Table 7.5. The *LFU* strategy was the only requiring the 0.9 threshold this time. *LRU* was again the only one staying under the memory usage limit all the time. The amount of lost violations in is again high for all strategies with *LRU* being the best again, tiny bit before *Random*.

Strategy	Threshold	Runtime (s)	Max memory usage (MiB)	Violations diff (%)
Random	0.7	95.17	1038.52	34.9
LRU	0.7	96.99	1011.22	31.5
LFU	0.9	91.67	1072.37	49.42

Table 7.5: Comparison of GC strategies when applied to the property complexity scaling TC (*NOoOE* monitoring mode).

Figure 7.8 confirms these trends. *Random* achieved substantial memory reduction, but *LRU* remained the most consistent overall.

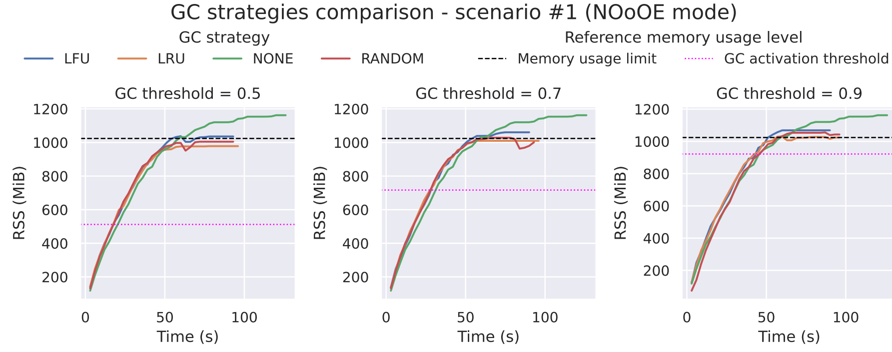


Figure 7.8: Visualization of the memory usages when different GC strategies are applied to the TC with complex properties (*NOoOE* monitoring mode).

Evaluation for Scenario with High Number of Properties Without Event Sharing

Standard Monitoring Mode

The baseline measurements without GC and the corresponding memory limit were as follows:

- **Base metrics without GC**: *runtime*=155.75 s, *memory usage*=2069.97 MiB
- **Max memory usage limit**: 1811

As shown in Table 7.6, the best threshold was 0.7. All strategies stayed under the memory limit. *LFU* again showed the highest violation loss.

Strategy	Threshold	Runtime (s)	Max memory usage (MiB)	Violations diff (%)
Random	0.7	135.81	1776.34	25.19
LRU	0.7	133.27	1759.53	20.31
LFU	0.7	122.9	1748.0	38.13

Table 7.6: Comparison of GC strategies when applied to the property number scaling without event sharing TC (*Standard* monitoring mode).

Figure 7.9 illustrates that all strategies kept peak memory near the configured limit, especially at thresholds 0.7 and 0.9. Memory usage trends across strategies were nearly identical.

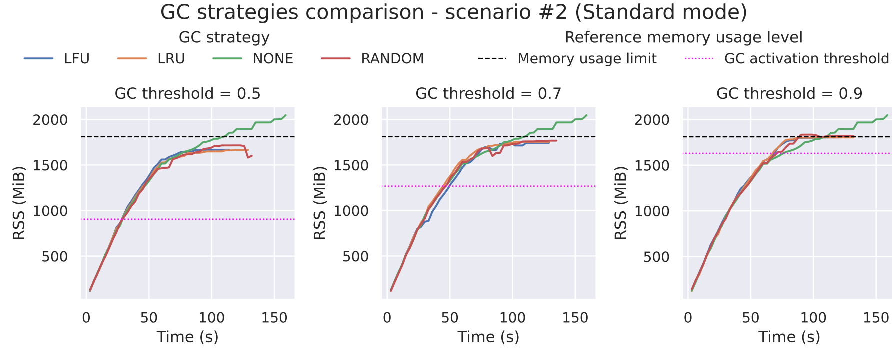


Figure 7.9: Visualization of the memory usages when different GC strategies are applied to the TC with high number of properties that do not share events (*Standard* monitoring mode).

No Out of Order Event Monitoring Mode

The baseline measurements without GC and the corresponding memory limit were as follows:

- **Base metrics without GC**: *runtime*=161.1 s, *memory usage*=2100.78 MiB
- **Max memory usage limit**: 1838

Results were similar to the standard mode (Table 7.7). Both *LRU* and *Random* performed well, with *LRU* slightly ahead.

Strategy	Threshold	Runtime (s)	Max memory usage (MiB)	Violations diff (%)
Random	0.7	139.1	1785.47	25.01
LRU	0.7	138.78	1767.59	20.46
LFU	0.7	132.21	1751.86	37.37

Table 7.7: Comparison of GC strategies when applied to the property number scaling without event sharing TC (*NOoOE* monitoring mode).

Memory usage plots (Figure 7.10) for 0.9 threshold were nearly identical. For 0.7, memory usage converged at the end, though trends varied. Drops in memory usage from the cleaning service were visible again, especially in the baseline.

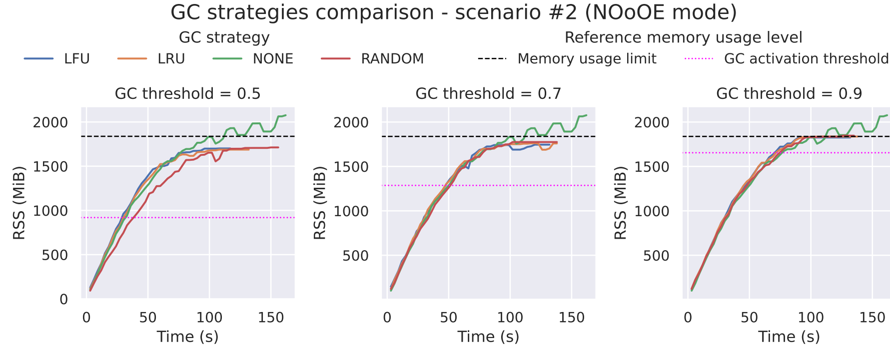


Figure 7.10: Visualization of the memory usages when different GC strategies are applied to the TC with high number of properties that do not share events (*NOoOE* monitoring mode).

Evaluation for Scenario with High Number of Properties Sharing Events

Standard Monitoring Mode

The baseline measurements without GC and the corresponding memory limit were as follows:

- **Base metrics without GC**: *runtime*=132.95 s, *memory usage*=1886.1 MiB
- **Max memory usage limit**: 1650

Table 7.8 indicates that lower threshold values were more effective in this scenario. The *Random* strategy achieved the fewest lost violations (approximately four times fewer than the other strategies), but slightly exceeded the memory limit.

Strategy	Threshold	Runtime (s)	Max memory usage (MiB)	Violations diff (%)
Random	0.5	125.89	1654.95	7.84
LRU	0.7	104.18	1517.94	27.58
LFU	0.7	106.47	1622.04	28.37

Table 7.8: Comparison of GC strategies when applied to the property number scaling with event sharing TC (*Standard* monitoring mode).

Figure 7.11 presents a different memory usage pattern—more linear than logarithmic. Strategies mostly tracked closely with the baseline. *Random* ran longer, but kept memory use high (even above the limit). Others ran faster, likely due to reduced workload resulting from suboptimal victim selection.

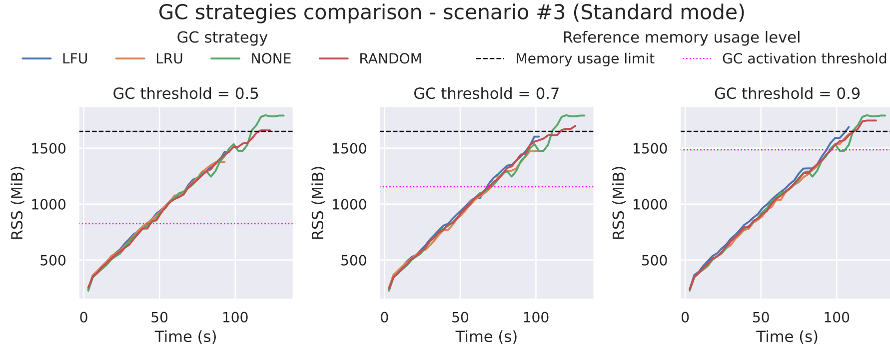


Figure 7.11: Visualization of the memory usages when different GC strategies are applied to the TC with high number of properties that share events (*Standard* monitoring mode).

No Out of Order Event Monitoring Mode

The baseline measurements without GC and the corresponding memory limit were as follows:

- **Base metrics without GC**: *runtime*=140.93 s, *memory usage*=1935.17 MiB
- **Max memory usage limit**: 1693

Results were nearly identical to standard mode (Table 7.9). *Random* again achieved the best performance and stayed within the memory limit this time.

Strategy	Threshold	Runtime (s)	Max memory usage (MiB)	Violations diff (%)
Random	0.5	137.41	1656.29	7.59
LRU	0.7	108.49	1586.32	27.11
LFU	0.7	111.74	1645.87	27.99

Table 7.9: Comparison of GC strategies when applied to the property number scaling with event sharing TC (*NOoOE* monitoring mode).

Figure 7.12 confirms this, showing *Random* performing better near the memory limit compared to the standard mode.

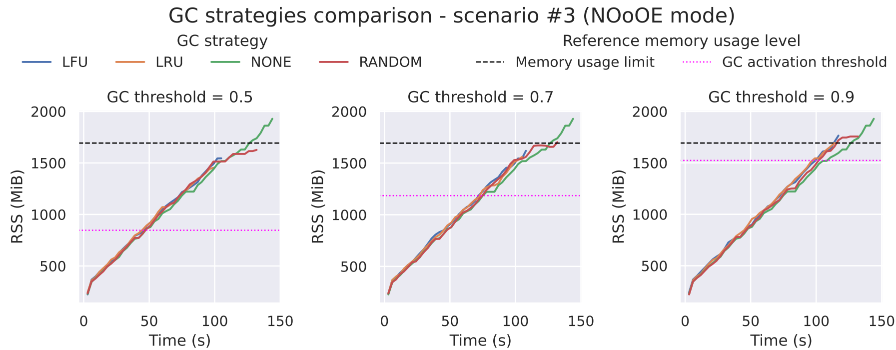


Figure 7.12: Visualization of the memory usages when different GC strategies are applied to the TC with high number of properties that share events (*NOoOE* monitoring mode).

Summary

The comparison of GC strategies revealed several key insights. Choosing a mid-range threshold value (e.g., around 0.7) proved to be the most generally effective—consistently delivering optimal or near-optimal results across all scenarios. Since GC inherently reduces workload by discarding monitor instances (and thus potentially losing violations), it also led to shorter runtimes. The effects of GC were largely consistent across both monitoring modes, with only minor differences observed.

Among the strategies, *LRU* performed best in most scenarios. It effectively reduced memory usage while minimizing information loss.

The *Random* strategy also proved to be a strong option, often coming close to *LRU*. In the scenario with multiple properties and event sharing, it outperformed the others. Deterministic strategies (i.e., *LRU* and *LFU*) tended to evict monitor instances related to shared events. This suboptimal choice was repeated across properties, leading to higher information loss. Random sampling, by selecting victims uniformly, avoided this issue.

Finally, *LFU* performed the worst in nearly all scenarios. Although it kept memory usage relatively close to the configured limit, it resulted in significantly higher information loss compared to the other strategies.

7.3 Functional Verification

In addition to evaluating the performance of *Plogchecker 3.0*, its functional correctness was thoroughly verified. This verification was carried out on multiple levels. Initially, unit testing was performed on all modules that comprise the tool. Subsequently, integration tests were designed to assess the overall behavior of the tool. Finally, the tool's compliance with the multi-platform usage requirement was verified.

Unit Tests

The unit tests were implemented using Go's built-in `testing`[3] package. The test suite was structured according to Go conventions—test files were placed in the same package as the code under test and named using the `_test.go` suffix.

Test coverage was analyzed using the `go-test-coverage`[4] tool. This tool offers a simple yet robust interface, making it well-suited for this purpose. It calculates coverage based on statement coverage and allows specification of minimum coverage thresholds at the file, package or global level. The thresholds and actual results are as follows:

- *Per-file threshold*: 90%
- *Per-package threshold*: 90%
- *Global threshold*: 95%
- **Actual global coverage achieved**: 99.3%

[3] It provides support for automated testing of Go programs—`https://pkg.go.dev/testing`.
[4] Tool designed for reporting the test coverage—`https://github.com/vladopajic/go-test-coverage`.

Integration Tests

Integration testing was conducted using a fully custom solution with an end-to-end testing approach. In each test case, *Plogchecker 3.0* is given input files and is expected to produce correct outputs. Each test case is defined by the following components:

- `config.yaml`: configuration file for *Plogchecker 3.0* specific to the test case.
- `events.txt`: input event stream for monitoring.
- `report.ndjson`: the expected output report containing detected violations.
- `test_config.txt`: additional test parameters (e.g., expected return code, extra CLI options).

A test is considered passed only if *Plogchecker 3.0* produces the correct NDJSON report file and returns the expected exit code. The order of entries in the NDJSON output must match the expected file exactly, although the order of fields within each JSON object may vary—JSON structural equivalence is enforced. Test definitions are stored in the `integration_tests/` directory. A shell script, `run_integration_tests.sh`, automates test execution, evaluation and result reporting.

The integration tests primarily focus on verifying the core functionality as defined by the requirements listed in Table 5.2. Additional test cases cover edge scenarios and negative conditions. For clarity and organization, the tests are grouped by functional area, with each group targeting specific area. A summary of the test groups and their corresponding requirements is presented in Table 7.10.

Test group	Covered requirements
`001_invalid_program_arguments`	
`002_valid_program_arguments`	1.4
`003_invalid_configuration_file`	
`004_parameter_constraints`	2.7, 2.8, 2.9, 2.10, 2.11
`005_property_operators`	2.5, 2.6, 2.12
`006_standard_monitoring_mode`	2.1, 2.2, 2.3, 2.4, 2.5, 2.6
`007_no_out_of_order_events_monitoring_mode`	2.1, 2.2, 2.3, 2.4, 2.5, 2.6

Table 7.10: Overview of the test groups used for the integration testing of *Plogchecker 3.0*. For each group, the corresponding list of verified requirements is provided.

Cross-platform Usage

As required by Requirement 1.3, *Plogchecker 3.0* must support cross-platform compatibility. To validate this, the tool was compiled and tested on a variety of platforms with different operating systems and architectures. A summary of the tested platforms is provided in Table 7.11, which includes OS version, architecture and other relevant system details.

For each platform, successful compilation and execution of the full unit test suite were confirmed. All platforms passed these checks, demonstrating that *Plogchecker 3.0* meets the portability requirement.

OS	**Arch**	**Kernel / Build**	**Notes**
Windows 11 (24H2)	x86_64	26100.4652	Native
macOS Sonoma 14.7.6	arm64	Darwin 23.7.0	Native on Apple M2
Ubuntu 22.04.5	x86_64	6.6.87.2-microsoft-standard-WSL2	Via WSL2 on Win 11

Table 7.11: Specification of the platforms on which *Plogchecker 3.0* was tested.

Chapter 8

Conclusion

The initial part of this thesis introduced the fundamental concepts of parametric runtime verification, examining various implementation approaches with a particular focus on *trace slicing*. It also provided formal foundations for property specification and the generation of monitors based on such specifications. Additionally, a brief overview of several state-of-the-art tools utilizing *trace slicing* was presented, emphasizing their monitoring algorithms, optimizations and resource management strategies.

The primary objective of this thesis was to reimplement and optimize an existing tool for parametric runtime verification—*Plogchecker 2.0*. Consequently, the following chapter analyzed the tool's current state and outlined areas for improvement. The design, strengths and limitations were critically examined, with particular attention to the deficiencies of its monitoring algorithm.

An enhanced version, *Plogchecker 3.0*, was proposed with a focus on improving scalability through a redesigned monitoring algorithm. A key innovation was the introduction of indexing tree structures, enabling efficient retrieval of monitor instances associated with specific parameter combinations. The new algorithm was designed in a generic manner to support multiple monitoring modes. Currently, two modes have been introduced—*Standard* and *No Out-of-Order Events* (*NOoOE*). These modes primarily differ in the strictness of the rules applied when updating monitor instances in response to incoming events. Additionally, improvements were made to the garbage collection process, allowing the tool not only to clean up obsolete monitor instances but also to preemptively discard incomplete ones, thereby avoiding out-of-memory crashes. To further enhance expressiveness, several new data types and operators were also introduced.

Subsequently, *Plogchecker 3.0* was implemented according to the proposed design. The tool comprises several independent, concurrently running services that collectively perform the monitoring process. While some services handle core monitoring tasks (such as event processing and monitor instance management), others focus on resource efficiency (e.g., pruning indexing trees and performing preemptive garbage collection).

The implementation was thoroughly evaluated through a combination of functional and performance tests. Functional correctness was verified using both unit and integration testing, demonstrating that *Plogchecker 3.0* satisfies all specified functional and non-functional requirements. Cross-platform compatibility was also successfully confirmed.

The first part of the performance evaluation focused on scalability with respect to various factors. Tests were conducted independently for both monitoring modes. Results showed that the memory usage of the two modes was nearly identical, with differences below 2% across all scenarios. However, runtime performance varied significantly—the *NOoOE* mode

was up to 10% slower than the *Standard* mode, primarily due to stricter processing logic required during monitoring.

Benchmark results confirmed that *Plogchecker 3.0* scales well with an increasing number or complexity of monitored properties, exhibiting near-linear growth in both runtime and memory usage. Regarding scalability with respect to the number of parameters per property, performance remained linear up to eight parameters. Beyond this point, both memory and runtime grew dramatically, likely due to state explosion in the number of monitor instances.

The second part of the performance evaluation compared the garbage collection (GC) strategies. The *LRU* strategy proved most effective, achieving the best trade-off between memory savings and preservation of violation detections. The *Random* strategy also performed well, with results close to *LRU* in most scenarios. In contrast, the *LFU* strategy consistently performed the worst, resulting in a significantly higher number of lost violations.

Although the current implementation of *Plogchecker 3.0* demonstrated strong performance and functionality, there remains room for future enhancements. One promising direction is further scaling optimization, particularly beyond eight parameters per property. Garbage collection could also be refined—for instance, the current approach distributes eviction efforts proportionally across all monitored properties based on the number of monitor instances. However, this might not always be ideal, as some properties may be more critical than others. Introducing a priority system for properties could improve this. Moreover, new GC strategies could be added. Given that both LRU and Random perform well, a hybrid approach combining their strengths may yield even better results. Alternatively, adaptive strategies could be introduced—e.g., ones that dynamically adjust based on observed input event patterns. Lastly, extending the set of supported data types, operators and constraint functions could significantly improve the tool's expressiveness and usability.

All of the mentioned improvements represent valuable directions for future research, either in the form of subsequent theses or as part of ongoing work within the VeriFIT[1] Research Group at Brno University of Technology.

[1]Automated Analysis and Verification Research Group — VeriFIT (`https://www.fit.vut.cz/research/group/verifit/`)

Bibliography

[1] Allan, C.; Avgustinov, P.; Christensen, A. S.; Hendren, L.; Kuzins, S. et al. Adding trace matching with free variables to AspectJ. *ACM SIGPLAN Notices.* ACM New York, NY, USA, 2005, vol. 40, no. 10, p. 345–364.

[2] Avgustinov, P.; Tibble, J. and Moor, O. de. Making trace monitors feasible. In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications.* 2007, p. 589–608.

[3] Basin, D.; Harvan, M.; Klaedtke, F. and Zălinescu, E. MONPOLY: Monitoring usage-control policies. In: Springer. *Runtime Verification: Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers 2.* 2012, p. 360–364.

[4] Chen, F. and Roşu, G. Parametric trace slicing and monitoring. In: Springer. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* 2009, p. 246–261.

[5] Colombo, C.; Pace, G. J. and Schneider, G. Dynamic event-based runtime monitoring of real-time and contextual properties. In: Springer. *Formal Methods for Industrial Critical Systems: 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers 13.* 2009, p. 135–149.

[6] Colombo, C.; Pace, G. J. and Schneider, G. LARVA—safer monitoring of real-time java programs (Tool Paper). In: IEEE. *2009 seventh ieee international conference on software engineering and formal methods.* 2009, p. 33–37.

[7] d'Angelo, B.; Sankaranarayanan, S.; Sánchez, C.; Robinson, W.; Finkbeiner, B. et al. LOLA: runtime monitoring of synchronous systems. In: IEEE. *12th International Symposium on Temporal Representation and Reasoning (TIME'05).* 2005, p. 166–174.

[8] Decker, N.; Leucker, M. and Thoma, D. Monitoring modulo theories. *International Journal on Software Tools for Technology Transfer.* Springer, 2016, vol. 18, p. 205–225.

[9] Falcone, Y.; Havelund, K. and Reger, G. A tutorial on runtime verification. *Engineering dependable software systems.* IOS Press, 2013, p. 141–175.

[10] Galil, Z. and Italiano, G. F. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys (CSUR).* ACM New York, NY, USA, 1991, vol. 23, no. 3, p. 319–344.

[11] Havelund, K. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer*. Springer, 2015, vol. 17, p. 143–170.

[12] Havelund, K.; Peled, D. and Ulus, D. First-order temporal logic monitoring with BDDs. *Formal Methods in System Design*. Springer, 2020, vol. 56, no. 1, p. 1–21.

[13] Havelund, K.; Reger, G.; Thoma, D. and Zălinescu, E. Monitoring events that carry data. *Lectures on Runtime Verification: Introductory and Advanced Topics*. Springer, 2018, p. 61–102.

[14] Hopcroft, J. An n log n algorithm for minimizing states in a finite automaton. In: *Theory of machines and computations*. Elsevier, 1971, p. 189–196.

[15] Hopcroft, J. E.; Motwani, R. and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. Pearson Education, 2007. ISBN 0-321-45536-3.

[16] Jin, D. *Making runtime monitoring of parametric properties practical*. 2012. Dissertation. University of Illinois at Urbana-Champaign.

[17] Jin, D.; Meredith, P. O.; Griffith, D. and Rosu, G. Garbage collection for monitoring parametric properties. *ACM SIGPLAN Notices*. ACM New York, NY, USA, 2011, vol. 46, no. 6, p. 415–424.

[18] Luo, Q.; Zhang, Y.; Lee, C.; Jin, D.; Meredith, P. O. et al. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In: Springer. *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5*. 2014, p. 285–300.

[19] Meredith, P. O.; Jin, D.; Griffith, D.; Chen, F. and Roşu, G. An overview of the MOP runtime verification framework. *International Journal on Software Tools for Technology Transfer*. Springer, 2012, vol. 14, no. 3, p. 249–289.

[20] Moore, E. F. et al. Gedanken-experiments on sequential machines. *Automata studies*. Princeton, 1956, vol. 34, p. 129–153.

[21] Mutňanský, F. *Ověřování parametrických vlastností nad záznamy běhů programů*. Brno, CZ, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/22424/.

[22] Sipser, M. *Introduction to the Theory of Computation*. 3rd ed. Cengage Learning, 2013. ISBN 1-133-18779-X.

[23] The IEEE and The Open Group. *Regular Expressions* online. 2024. Available at: https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/V1_chap09.html. [cit. 2024-12-10].

[24] Čaládi, F. *Ověřování parametrických vlastností nad záznamy běhů programů*. Brno, CZ, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/23298/.

Appendix A

Contents of the Attached Memory Media

This appendix lists the contents of the attached memory medium. It includes the following items:

- `plogchecker/`: source code of *Plogchecker 3.0* (refer to the `README.md` file in this folder for further guidance)
- `thesis-source/`: source files of the thesis
- `thesis_xsuran07.pdf`: the thesis in PDF format

Appendix B

Semantic Checks for Binary Operators in Constraints

This appendix presents the semantic rules for binary operators used by *Plogchecker 3.0* when evaluating expressions representing constraints. These rules are summarized in Table B.1. For each binary operator, the table specifies the allowed combinations of operand types as well as the resulting type.

Operator	**Left Operand Type**	**Right Operand Type**	**Result Type**
PLUS (+)	NUMBER	NUMBER	NUMBER
	WORD	WORD	WORD
	DATE	DURATION	DATE
	DURATION	DATE	DATE
PLUS (-)	NUMBER	NUMBER	NUMBER
	DATE	DURATION	DATE
	DATE	DATE	DURATION
GE (>=)	NUMBER	NUMBER	BOOL
	DATE	DATE	BOOL
	DURATION	DURATION	BOOL
GT (>)	NUMBER	NUMBER	BOOL
	DATE	DATE	BOOL
	DURATION	DURATION	BOOL
LE (<=)	NUMBER	NUMBER	BOOL
	DATE	DATE	BOOL
	DURATION	DURATION	BOOL
LT (<)	NUMBER	NUMBER	BOOL
	DATE	DATE	BOOL
	DURATION	DURATION	BOOL
EQ (=)	NUMBER	NUMBER	BOOL
	DATE	DATE	BOOL
	WORD	WORD	BOOL
	BOOL	BOOL	BOOL
	DURATION	DURATION	BOOL
	PATH	PATH	BOOL
	IP	IP	BOOL
NEQ (!=)	NUMBER	NUMBER	BOOL
	DATE	DATE	BOOL
	WORD	WORD	BOOL
	BOOL	BOOL	BOOL
	DURATION	DURATION	BOOL
	PATH	PATH	BOOL
	IP	IP	BOOL

Table B.1: Rules for binary operators used by *Plogchecker 3.0* during the semantic analysis of constraints.

Appendix C

Full Results of the Garbage Collection Strategies Comparison

This appendix presents the detailed results of the comparison of garbage collection (GC) strategies, as discussed in Section 7.2. The comparison was carried out across four distinct scenarios. For each scenario, the evaluation was conducted separately for both monitoring modes. Every GC strategy was tested using three different threshold values—0.5, 0.7 and 0.9. The results from all these runs, across all scenarios, are presented in the sections below.

Evaluations for the Scenario with a High Number of Parameters

This section presents the results from the scenario with a high number of parameters. The results for the *Standard* monitoring mode are shown in Table C.1. Conversely, the results for the *No Out-of-Order Event* (*NOoOE*) monitoring mode are presented in Table C.2.

Threshold	**Strategy**	**Runtime (s)**	**Max memory usage (MiB)**	**Violations diff (%)**
0.5	*Random*	268.59	1274.52	22.34
	LRU	277.38	1317.15	16.08
	LFU	248.75	1251.89	24.79
0.7	*Random*	293.96	1357.99	15.8
	LRU	298.14	1382.65	8.6
	LFU	271.28	1332.35	18.2
0.9	*Random*	304.56	1428.36	10.48
	LRU	315.1	1428.09	3.88
	LFU	285.96	1411.66	12.93

Table C.1: Results of applying GC strategies to the scenario with a high number of parameters (*Standard* monitoring mode).

Threshold	Strategy	Runtime (s)	Max memory usage (MiB)	Violations diff (%)
0.5	*Random*	293.77	1259.08	23.39
	LRU	355.4	1279.59	18.74
	LFU	292.54	1227.16	25.95
0.7	*Random*	313.25	1354.56	16.2
	LRU	323.86	1379.97	8.89
	LFU	292.59	1325.22	18.44
0.9	*Random*	331.19	1416.97	10.4
	LRU	341.63	1425.19	4.04
	LFU	311.67	1402.64	12.99

Table C.2: Results of applying GC strategies to the scenario with a high number of parameters (*NOoOE* monitoring mode).

Evaluation for the Scenario with Complex Property Patterns

This section presents the evaluation results for the scenario involving complex property patterns. The outcomes for the *Standard* monitoring mode are shown in Table C.3, while the results for the *NOoOE* monitoring mode are provided in Table C.4.

Threshold	Strategy	Runtime (s)	Max memory usage (MiB)	Violations diff (%)
0.5	*Random*	99.88	1012.86	36.93
	LRU	92.55	980.28	31.97
	LFU	93.46	1116.97	50.18
0.7	*Random*	93.46	1053.27	35.79
	LRU	94.28	1004.97	29.64
	LFU	93.56	1111.52	49.23
0.9	*Random*	93.59	1064.95	33.99
	LRU	91.84	1038.29	30.77
	LFU	96.82	1114.54	46.27

Table C.3: Results of applying GC strategies in the scenario with complex property patterns (*Standard* monitoring mode).

Threshold	Strategy	Runtime (s)	Max memory usage (MiB)	Violations diff (%)
0.5	*Random*	95.07	1013.51	35.77
	LRU	97.45	982.02	33.56
	LFU	92.64	1051.14	53.5
0.7	*Random*	95.17	1038.52	34.9
	LRU	96.99	1011.22	31.5
	LFU	90.95	1068.15	51.3
0.9	*Random*	96.31	1064.87	34.74
	LRU	101.87	1040.54	30.21
	LFU	91.67	1072.37	49.42

Table C.4: Results of applying GC strategies in the scenario with complex property patterns (*NOoOE* monitoring mode).

Evaluation for Scenario with High Number of Properties Without Event Sharing

This section provides the evaluation results for the scenario characterized by a high number of properties without event sharing. The results for the *Standard* monitoring mode are displayed in Table C.5, while those for the *NOoOE* monitoring mode appear in Table C.6.

Threshold	Strategy	Runtime (s)	Max memory usage (MiB)	Violations diff (%)
0.5	*Random*	133.27	1720.74	28.99
	LRU	130.26	1673.92	25.31
	LFU	121.25	1678.28	42.21
0.7	*Random*	135.81	1776.34	25.19
	LRU	133.27	1759.53	20.31
	LFU	122.9	1748.0	38.13
0.9	*Random*	134.7	1860.17	25.2
	LRU	133.75	1828.57	19.84
	LFU	125.55	1820.4	36.65

Table C.5: Results of applying GC strategies in the scenario with a high number of non-overlapping properties (*Standard* monitoring mode).

Threshold	Strategy	Runtime (s)	Max memory usage (MiB)	Violations diff (%)
0.5	*Random*	145.16	1721.9	30.26
	LRU	134.99	1694.84	25.42
	LFU	123.42	1702.71	41.74
0.7	*Random*	139.1	1785.47	25.01
	LRU	138.78	1767.59	20.46
	LFU	132.21	1751.86	37.37
0.9	*Random*	138.83	1872.3	23.05
	LRU	142.11	1844.51	18.95
	LFU	132.4	1841.64	35.51

Table C.6: Results of applying GC strategies in the scenario with a high number of non-overlapping properties (*NOoOE* monitoring mode).

Evaluation for Scenario with High Number of Properties Sharing Events

This section presents the evaluation results for the scenario involving a high number of properties with overlapping event usage. The outcomes for the *Standard* monitoring mode are shown in Table C.7, while those for the *NOoOE* monitoring mode are provided in Table C.8.

Threshold	Strategy	Runtime (s)	Max memory usage (MiB)	Violations diff (%)
0.5	*Random*	125.89	1654.95	7.84
	LRU	94.62	1382.75	33.93
	LFU	97.31	1501.1	35.91
0.7	*Random*	126.56	1704.18	6.02
	LRU	104.18	1517.94	27.58
	LFU	106.47	1622.04	28.37
0.9	*Random*	127.89	1752.63	4.4
	LRU	111.18	1674.28	20.22
	LFU	110.61	1705.02	21.11

Table C.7: Performance of GC strategies in the scenario with a high number of properties sharing events (*Standard* monitoring mode).

Threshold	Strategy	Runtime (s)	Max memory usage (MiB)	Violations diff (%)
0.5	*Random*	137.41	1656.29	7.59
	LRU	100.67	1475.81	33.97
	LFU	106.16	1559.75	34.85
0.7	*Random*	132.35	1713.27	5.14
	LRU	108.49	1586.32	27.11
	LFU	111.74	1645.87	27.99
0.9	*Random*	139.73	1770.51	3.51
	LRU	121.58	1750.44	17.18
	LFU	117.14	1772.13	19.32

Table C.8: Performance of GC strategies in the scenario with a high number of properties sharing events (*NOoOE* monitoring mode).