BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PARAMETRIC PROPERTIES FOR LOG CHECKER
OVĚŘOVÁNÍ PARAMETRICKÝCH VLASTNOSTÍ NAD ZÁZNAMY BĚHŮ PROGRAMŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. FILIP ČALÁDI**
AUTOR PRÁCE

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

BRNO 2022

Ústav inteligentních systémů (UITS) Akademický rok 2021/2022

Zadání diplomové práce

23298

Student:	**Čaládi Filip, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Verifikace a testování software
Název:	**Ověřování parametrických vlastností nad záznamy běhů programů** **Parametric Properties for Log Checker**
Kategorie:	Analýza a testování softwaru

Zadání:

1. Seznamte se s metodami verifikace programů za běhu. Nastudujte nástroj plogchecker z platformy Testos. Nastudujte možnosti specifikace sekvence událostí založených na regulárních výrazech nebo na výrazech temporálních logik.
2. Navrhněte jazyk pro specifikaci parametrických sekvencí událostí. Navrhněte rozšíření nástroje plogchecker, které umožní kontrolovat splnění či porušení parametrických sekvencí událostí. Rozšíření zaměřte na podporu různých datových typů parametrů (např. řetězce, čísla, datum a čas).
3. Implementujte rozšíření jako refaktorovaný kód nástroje plogchecker.
4. Ověřte funkcionalitu nového nástroje pomocí automatizovaných testů základní funkcionality.

Literatura:

- Klaus Havelund, Giles Reger, Daniel Thoma, Eugen Zalinescu: Monitoring Events that Carry Data. Lectures on Runtime Verification 2018: 61-102.
- Domovská stránka projektu plogchecker, url: https://pajda.fit.vutbr.cz/testos/plogchecker
- Domovská stránka projektu Kint, url: https://kint-php.github.io/kint/
- Modul trace pro Python 2.x, url: https://docs.python.org/2/library/trace.html

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1. listopadu 2021
Datum odevzdání:	18. května 2022
Datum schválení:	3. listopadu 2021

Abstract

Plogchecker 2.0 is a tool for verification of user-defined properties over sequences of events in the traces of the program. The implementation of this tool mainly builds on the previous version of the tool *Plogchecker*. The main idea behind these tools is that the user has to specify system properties (parametric or non-parametric), make any program run records available to the verification tool and let the tool analyze. The analysis output is the report about the violation of specified properties with sequences of events that caused the error. This thesis proposes a new algorithm that optimizes the processing of event sequences against user-defined properties specifications. The optimizations are focused on both *scalability* as well as *precision*. Furthermore, it adds support for various parametric data types, such as *string*, *number*, *date* and *time*. Finally, it offers an easier and more comfortable way to specify such parametric properties. Throughout the series of experiments, it is shown that *Plogchecker 2.0* is more scalable and precise compared to previous version of *Plogchecker*.

Abstrakt

Plogchecker 2.0 je nástroj zameraný na verifikáciu užívatelom definovaných vlastností nad sekvenciou udalostí generovaných programom. Implementácia tohoto nástroja stavá hlavne na už implementovanom nástroji *Plogchecker*. Hlavná mýšlienka týchto nástrojov je, že užívatel musí špecifikovať želané vlastnosti (parametrické alebo neparametrické), sprístupniť záznam behu programu verifikačnému nástroju a konečne prenechať analýzu na tento nástroj. Výstup analýzy je report o porušení špecifikovaných vlastností spolu so sekvenciami udalostí, ktoré spôsobili chybu. Táto práca predstavuje nový algoritmus , ktorý optimalizuje spracovanie sekvenie udalostí nad užívatelom definovanými vlastnosťami. Táto optimalizácia sa zameriava ako na *škálovatelnosť* tak aj *presnosť*. Ďalej, je pridaná podpora pre rôzne dátové typy parametrov, ako napríklad *reťazec*, *číslo*, *dátum* a *čas*. Nakoniec, táto práca ponúka jednoduchší a pohodlnejší spôsob vytvárania parametických vlastností. Počas experimentovania bolo ukázané, že *Plogchecker 2.0* je schopný väčšej škálovatelnosti a presnosti.

Keywords

runtime verification, parametric properties, non-parametric properties, extended regular expressions, finite state automatons, golang, grok, garbage collecting

Klíčová slova

verifikácia za behu, parametrické vlastnosti, neparametrické vlastnosti, rozšírené regulárne výrazy, konečné automaty, golang, grok, garbage collecting

Reference

ČALÁDI, Filip. *Parametric Properties for Log Checker.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, Ph.D.

Rozšířený abstrakt

Softwarové chyby sa v počítačových programoch vyskytujú už od samotného vzniku programovania. Vývojari majú k dispozícii mnoho spôsobov, ako odhalovať tieto chyby už v samotnej fáze vývoja software. Táto práca sa zaoberá jednou konkrétnou technikou testovania software, a to *verifikáciou za behu* ("runtime verification"). Ide o techniku testovania, v ktorej sa porovnáva *správanie* verifikovaného systému proti užívateľom definovaným *vlastnostiam.* Na základe takýchto dvoch vstupov, je potom nástroj, ktorý implementuje túto techniku verifikovania, schopný rozhodnúť či sledovaný systém spĺňa definované vlastnosti alebo nie. V rámci špecifikovania vlastností je každá vlastnosť štandardne definovaná sekvenciou udalostí. Ďalej, v rámci verifikácie za behu, udalosti typicky rozdeľujeme na takzvane *parametrické* a *neparametrické.* Parametrická udalosť je taká udalosť, ktorá v sebe nesie nejaká hodnotu identifikovanú jej parametrom. Napríklad, *"otvoril sa súbor s názvom f1".* Oproti tomu, neparametrická vlastnosť v sebe nenesie žiadnu hodnotu parametru. Napríklad, *"otvoril sa súbor"*, ale už nevieme s akým názvom.

V súčastnosti existuje mnoho nástrojov, ktoré implementujú techniku verifikácie za behu. Ich problémom však je, že väčšina z nich je vytvorená pre nejakú konkrétnu doménu. Napríklad, programovací jazyk, v ktorom, musí byť verifikovaný systém implementovaný. Alebo, že konkrétny nástroj implementuje podporu výlučne len pre parametrické alebo neparametrické udalosti, ale už nie jej kombináciu.

Jedným z nástrojov pre verifikáciu za behu je nástroj *Plogchecker.* Tento nástroj bol, pred pár rokmi, implementovaný v rámci diplomovej práce a výskumnej skupiny Testos na VUT FIT v Brne. Je potrebné poznamenať, že *Plogchecker* funguje pomerne dobre, ako prototyp (na demonštračné účely), no v súčastnosti nie je nasaditeľný na verifikáciu reálnych systémov. Za jeho hlavné nedostatky možno považovať pomerne nepohldnú špecifikáciu monitorovacích vlastností a ich udalostí. Za ďalší nedostatok možno považovať režijné náklady vygerované počas behu tohoto nástroja, špecificky teda *výpočetný čas* a množstvo *spotrebovanej pamäte.*

Táto diplomová práca predstavuje druhú generáciu už spomínaneho nástroja, a to *Plogchecker 2.0.* Tento nástroj sa snaží eliminovať spomínané nedostatky jeho predchodcu. Či už z pohľadu pohodlnejšieho špecifikovania monitorovacích vlastností alebo režijných nákladov. Napríklad, užívateľ je schopný v rámci špecifikovania parametrických vlastností definovať dátové typy jednotlivých parametrov. Medzi podporované dátové typy patria napríklad *reťazec*, *číslo* alebo *čas.* *Plogchecker 2.0* taktiež implementuje vylepšený monitorovací algoritmus, ktorého úlohou je zefektívniť ukladanie a monitorovanie prichádzajúcich udalostí verifikovaného systému.

Všetky navrhnuté a implementované rozšírenia a vylepšenia v rámci nástroja *Plogchecker 2.0* boli riadne odtestované. Ďalej bolo vykonané porovnanie výkonnosti nástrojov *Plogchecker* a *Plogchecker 2.0* z pohľadu *výpočetného času* a *spotrebovanej pamäte.* Testy ukázali, že v oboch prípadoch je *Plogchecker 2.0* na tom výkonnostne lepšie ako jeho predchodca.

Očakáva sa, že práca na tomto projekte bude aj naďalej pokračovať v rámci skupiny Testos na VUT FIT v Brne.

Parametric Properties for Log Checker

Declaration

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

....
Filip Čaládi
May 16, 2022

Acknowledgements

I would like to thank Ing. Aleš Smrčka, Ph.D., for the constant guidance and help he provided me during the whole year of my work on this thesis. I would also like to thank my family and friends for their patience and support.

Contents

1 Introduction **3**

2 Preliminaries **5**

2.1 Runtime Verification Taxonomy . . . 5
2.2 Background theory on Runtime Verification . . . 6
2.3 Specifying system properties . . . 8
2.4 Generating Monitors . . . 9

3 Baseline – Plogchecker **13**

3.1 Data-Flow Structure . . . 13
3.2 Property Specification . . . 13
3.3 Monitoring Algorithm . . . 15
3.4 Plogchecker Benchmark . . . 16

4 Proposal of Enhancements in Plogchecker 2.0 **22**

4.1 Monitoring algorithm . . . 22
4.2 Specification of Parametric Properties . . . 29
4.3 Discarding Monitored Sequences . . . 30
4.4 Garbage Collecting . . . 31
4.5 Implementation Language . . . 33

5 Implementation Details of Plogchecker 2.0 **35**

5.1 Property File . . . 35
5.2 Constraints Section . . . 38
5.3 Creating Tables from Defined Properties . . . 41
5.4 Monitoring Process . . . 54

6 Experiments and Evaluation **56**

6.1 Benchmark . . . 56
6.2 Functionality Testing . . . 58
6.3 Cross-Platform Compilation . . . 60

7 Conclusion **61**

Bibliography **63**

A Grammar Within the Constraints Section **65**

B Memory Consumption of Individual Benchmark Test Cases **67**

C Contents of the Attached Memory Media **73**

D Installation and User Manual **74**

Chapter 1

Introduction

Runtime verification is a computer science method that aims to extract the information from the software or hardware execution and use this information to check the correctness of the running system against some specific defined properties. However, many more techniques aim to check the correctness of some particular system, i.e., *model checking*, *type checking*, or *theorem proving*. While *model checking*, for example, is a powerful technique for software verification, even this method has its disadvantages. Many of those disadvantages are related to the process of model building. Consider a case where building the model can be *expensive* or *complex*. Building the model is not always possible as well. This can be true when the verified system needs to be viewed as a blackbox or part of the verified software is third-party or closed source. Those are the cases where *runtime verification* techniques can become helpful, potentially as lightweight verification methods instead of model checking, in checking the correctness of some particular system.

In recent years, many studies on *runtime verification* were focused on either synthesizing monitors from some high-level property specifications [6] or how to effectively process the information from system execution by the specific runtime monitor [10]. In its simplest form, a *monitor* is a verification software used to decide the correctness of any system execution against specified property or properties. The property can be essentially specified in any high-level specification language, i.e., *temporal logic* or *regular expressions* (or extended regular expressions). The monitor's output is then *yes/true* (correct behavior), *no/false* (incorrect behavior), or sometimes there is no clear verdict available, and the result would be *don't know* [9]. One can find a basic model of such a *runtime verification* tool in Figure 1.1.

This thesis aims to study, mainly, the problem of how to effectively process the information from system execution by the specific runtime monitor. All proposed ideas and algorithms are implemented within the *Plogchecker 2.0 tool*, which is practically a reimplementation of the previous version of this tool, *Plogchecker* [12]. Furthermore, both of them are developed within the *Testos*[1] group, which aims to create a modular platform of testing tools. One of the many challenges of this thesis was to create a *processing algorithm* that would be more effective and scalable than the one from *Plogchecker*. Such an algorithm is presented in Section 4.1. Moreover, this thesis adds support for various parametric data types within property specifications, such as string, number, date, and time. One can find more on this topic in Section 4.2.

[1]http://testos.org/

The rest of the thesis is organized as follows. Chapter 2 introduces basic terminology used within *runtime verification*. The original version of *Plogchecker*, its fundamentals, and structure are described in Chapter 3. Chapter 4 sums up all proposed enhancements implemented in *Plogchecker 2.0*. Next, Chapter 5 describes the implementation details of the most interesting parts of *Plogcchecker 2.0*. Chapter 6 then describes the functionality testing and performance comparison between *Plogcchecker* and *Plogchecker 2.0*. And finally, Chapter 7 concludes this thesis.

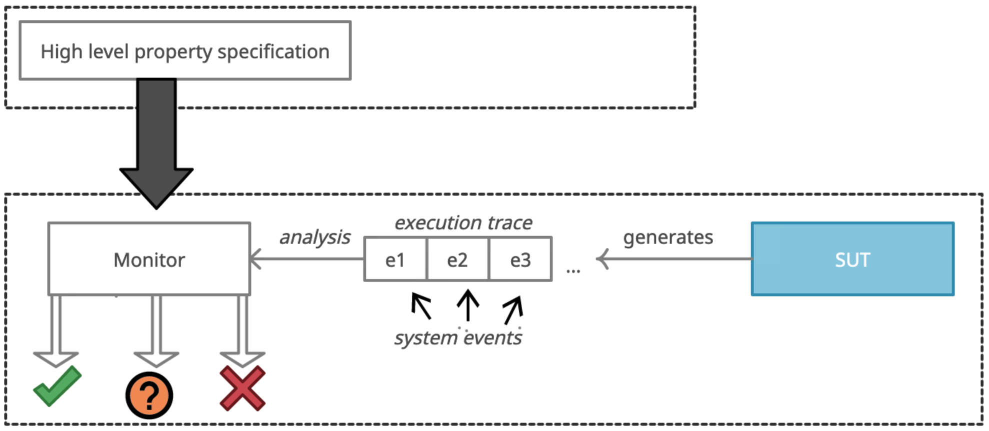


Figure 1.1: Basic model of *runtime verification* tool.

Chapter 2

Preliminaries

This chapter firstly defines some basic notions commonly used within *runtime verification* (event, trace, and property) in Sections 2.1 and 2.2. Section 2.3 talks about possible ways to specify some system properties. And to continue from that, Section 2.4 describes the process of yielding some specific monitors from user-defined properties.

2.1 Runtime Verification Taxonomy

Runtime verification can be helpful in many areas. One can find the division into several of them in Figure 2.1. The figure highlights four main areas; however, many more exists. The rest of this section is inspired by the conference paper, *Teaching runtime verificattion* [9].

Trace. *Runtime verification* may work on finite, finite expanding, or infinite traces (more on a trace in Section 2.2). In the case of *finite trace*, it can be understood as the record of the already terminated system run, which should be used for verification. In this case, the monitor can yield two verdicts, yes, meaning that the current behavior fulfills the correctness property, or no, meaning that the monitor found the misbehavior. *Finite expanding* traces are practically prefixes of *infinite traces.* In this case, two maxims need to be considered: *impartiality* and *anticipation. Impartiality* talks about the fact that the monitor can not evaluate a finite trace as true or false if some infinite trace exists, and the trace might lead to another verdict. On the other hand, *anticipation* talks about the fact that if every continuation of a *finite trace* leads to the same verdict, then the *finite trace* is evaluated to the same verdict. One can easily understand that the first maxim requires three different truth values: yes, no, and don't know, as, in some states of verification, the monitor does not know if it should evaluate to yes or no because it does not receive the full trace yet.

Stage. In the case of *offline monitoring*, we indirectly talk about *finite trace* as well. After the system execution, the verification is performed with all access to the complete system run record. On the other hand, *online monitoring* is performed along the system execution run. Thus we can say that the monitor is processing *finite expanding* execution trace.

Integration. An *inline monitor* is a piece of code that is directly merged with the verified system code. In comparison, an *outline monitor* is a separate program not directly affecting the verified system.

Application area. One can use *runtime verification* within different application areas, for example, *safety* or *security checking*. One can also use *runtime verification* to collect the information from the verified system, and use this information to, for example, measure its *performance*.

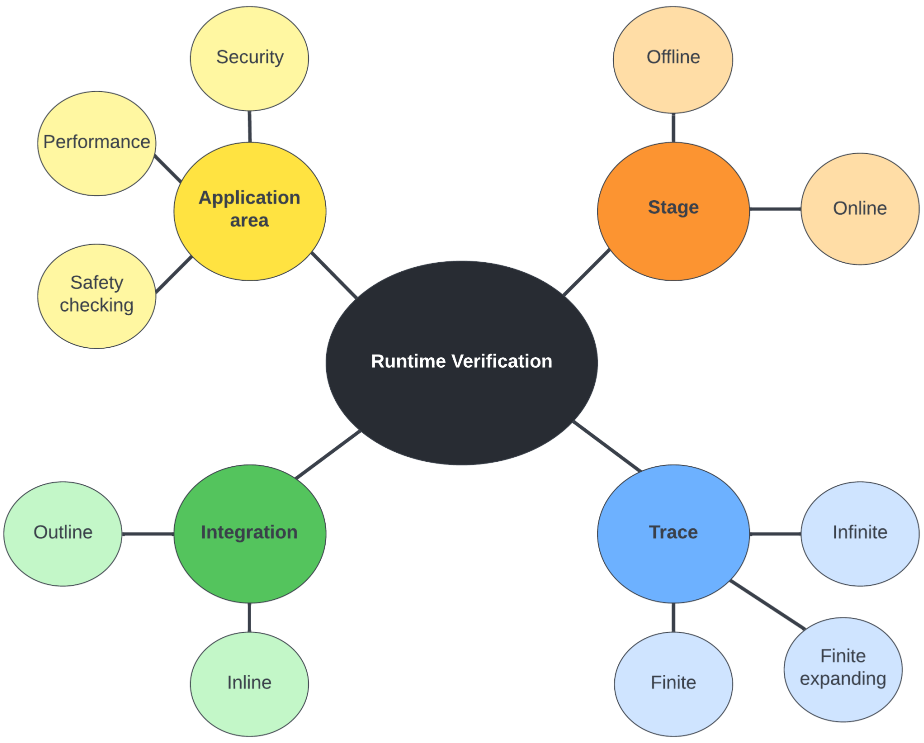


Figure 2.1: Taxonomy of *runtime verification*. Partially retrieved from [9].

In this thesis, both *Plogchecker* and *Plogchecker 2.0* tools operate on finite or finite expanding traces. Furthermore, they are capable of both online and offline monitoring. Finally, they both belong to the group of outline monitors, so monitoring is performed separately from the verified system.

2.2 Background theory on Runtime Verification

This section defines notions of trace, event, and property, all without parameters and then with parameters followed by examples. The section follows the notation and terminology from [3, 7] for consistency with other released papers from previous years.

Definition 2.2.1 (**Non-parametric event and trace**)**.** Let Σ be a set of events, called *base events*. Then *trace* is any finite sequence of events in Σ, that is an element in Σ^*. If event $e \in \Sigma$ appears in trace $w \in \Sigma^*$, it can be as well written like $e \in w$.

Example 2.2.1 (**Non-parametric event and trace**)**.** Consider a simple file object. This object can be opened, used, and finally closed. Then $\Sigma = \{open, use, close\}$. And execution traces corresponding to the usage of this file object are, for example, `'open.use.use.close'` or `'open.close'`.

Definition 2.2.2 (Non-parametric property). A non-parametric property P is a function $P : \Sigma^* \to C$ partitioning set of traces into categories C. Usually, C includes values such as *true*, *false* and *don't know*.

Example 2.2.2 (Non-parametric property). Consider a regular expression specification `open.use+.close`. Following the previous example, this specification states that once the file is opened, it has to be used at least once, and eventually, it has to be closed. Validating traces for such specification are, for example, `'open.use.use.close'` or `'open.use.close'`. Moreover, there are two more types of traces. The first one is not validating nor violating. It is a trace that can still lead to the validating state once the right sequence of events is received. For example, trace `'open.use'` in not validating nor violating. The second type of trace is the violating trace. The violating trace could be, for example, `'open.close'`. For such a trace, there is no chance that it will become validating in the future. Continuing from that, we can define set C as $C = \{true, false, don't\_know\}$. And for a given regular expression E, we can define it's associated property $P_E : \Sigma^* \to C$ as follows:

$$P_E(w) = \begin{cases} \text{true,} & \text{iff } w \in \Sigma^* \\ \text{false} & \text{iff } (w \notin \Sigma^*) \wedge (\forall w' \in \Sigma^* : ww' \notin \Sigma^*) \\ \text{don't\_know} & \text{otherwise} \end{cases}$$

Definition 2.2.3 (Parametric event and trace). Let X be a set of *parameters* and let V be the set of corresponding *parameter values*. Further, if Σ if set of base events from Definition 2.2.1 then let $\Sigma(X)$ be a set of corresponding parameter instances $e(\theta)$ where $e \in \Sigma$ and θ is partial function in $[X \rightharpoondown V]$. The *parametric trace* w_p is a trace with events in $\Sigma(\theta)$, or simply $w_p \in \Sigma(\theta)^*$.

Example 2.2.3 (Parametric event and trace). Continuing from above examples, a parametric trace can be, for example:

$$\textbf{open}(\theta_1)\textbf{.use}(\theta_1)\textbf{.open}(\theta_2)\textbf{.close}(\theta_1)\textbf{.close}(\theta_2)$$

Where θ_1 maps parameter `f` to value `f1` and θ_2 maps parameter `f` to value `f2`. With this in mind, the above trace can be written as:

$$\textbf{open}(f1)\textbf{.use}(f1)\textbf{.open}(f2)\textbf{.close}(f1)\textbf{.close}(f2)$$

This trace consists of two resources $f1$ and $f2$, where $f1$ was first opened, used exactly once, and finally closed. On the other hand, resource $f2$ was just opened and closed. Therefore no use action was recorded for resource $f2$.

Definition 2.2.4 (Parametric property). Lets consider set of parameters X together with their corresponding set of values V from Definition 2.2.3. Moreover, consider non-parametric property $P : \Sigma^* \to C$ from Definition 2.2.2. Then, the *parametric* property is defined as:

$$\Lambda\ X.P : \Sigma(X)^* \to [[X \rightharpoondown V] \to C]$$

A parametric property is therefore similar to a non-parametric property, except that the domain is parametric traces, and the output, rather than being one category, is a mapping of parameter instances to categories. This allows the parametric property to associate an output category for each parameter instance from $[X \rightharpoondown V]$.

2.3 Specifying system properties

One can generally specify the system properties in various formal languages. These range from, for example, *extended regular expressions* (ERE) [13] to query-oriented languages, such as *program query language* [11], and rule-based approaches, for example, *RuleR* [2]. Moreover, *temporal* logic-based languages are trendy in *runtime verification*, especially variants of *linear temporal logics* (LTL) [5].

Temporal properties can be divided into two main groups: *safety* and *liveness* properties, as something bad cannot happen, and something good will happen [8]. These are essentially monitorable by ERE as well. The obvious approach would be to divide monitored properties into good and bad properties. The *good* property stands for something that one would want to observe in the execution trace of a monitored system. Therefore this could be marked as *liveness* property. On the other hand, the *bad* property stands for something that one would not want to see in the execution trace of monitored property. Therefore this could be marked as a *safety* property. One can find the resulting verdicts after satisfaction or dissatisfaction of such properties in Table 2.1. This section further focuses on ERE, specifically their formal definition, as this formalism is directly used in both *Plogchecker* and *Plogchecker 2.0* tools.

	Good property verdict	**Bad property verdict**
satisfied	true	false
not satisfied	false	true

Table 2.1: Example of properties verdict after satisfaction or dissatisfaction.

Extended Regular Expressions

To explain ERE, firstly *regular expressions* (RE) need to be defined. Following [16], RE are defined in Definition 2.3.1.

Definition 2.3.1 (Regular expressions). Regular expressions over Σ are recursively defined as follows:

1. $\emptyset$ is a regular expression denoting the regular set $\emptyset$,
2. ϵ is a regular expression denoting the regular set $\{\epsilon\}$,
3. a is a regular expression denoting the regular set $\{a\}$ for all $a \in \Sigma$
4. if p, q are regular expression denoting the regular sets P and Q, then:
   - $(p + q)$ is regular expression denoting the regular set $P \cup Q$,
   - (pq) or $(p.q)$ is regular expression denoting the regular set $P \cdot Q$,

- (p^*) is regular expression denoting the regular set P^*.

5. There are no other regular expressions over Σ.

Continuing from that, now, ERE can be defined. The definition of ERE is given in Definition 2.3.2.

Definition 2.3.2. (**Extended regular expressions** [1]) ERE are expanding RE in the following way. They allow defining operators without the backslash. Furthermore, they contain the following operators:

1. ˆ matches the starting position within the string,
2. \$ matches the ending position of the string or the position just before a string-ending newline,
3. * matches the preceding element zero or more times,
4. + matches the preceding element one or more times,
5. ? matches the preceding element one or zero times,
6. | matches the preceding element or the following element,
7. $\{m, n\}$ matches the preceding element at least m and not more than n times,
8. $\{m\}$ matches the preceding element exactly m times,
9. $\{m, \}$ matches the preceding element at least m times,
10. $\{, n\}$ matches the preceding element not more than n times.

2.4 Generating Monitors

Generating monitors from some formal specification language is another crucial aspect of *runtime verification*. However, unlike in the decision process of what standard language to use when specifying system properties, there are limitations on what formal monitor model to use once one agrees on the specific formal property language.

For example, in the case of *LTL* properties, the general approach towards monitoring is that such properties are converted into an automaton model called *Büchi automaton* [5]. However, more focus has been on monitoring parametric properties in recent years. Since the monitor has to store observed properties and data they carry, the challenge is to use, potentially, more efficient monitor representation for manipulation over such properties. One of the approaches, for example, introduces a representation of *first-order past LTL* logic (which allows quantification over data occurring as parameters) with *binary decision diagrams* [5].

Both tools of interest, *Plogchecker* and *Plogchecker 2.0*, use RE as the language in which the system properties are specified. Thus, the simple and straightforward approach to generate runtime monitors is to use conversion from RE to *deterministic finite automaton* (DFA). And therefore, conversion from RE to DFA is further introduced and formally defined.

Conversion from RE to DFA

Following [16], the conversion from RE to DFA can be divided into two basic steps. Firstly, one needs to convert a given regular expression to the so-called *extended finite automaton* (EFA) that will be converted to DFA in the second step. Thus, EFA is formally defined in Definition 2.4.1. Moreover, the conversion algorithm from RE to EFA is described in Algorithm 2.4.1. Finally, conversion from EFA to DFA is introduced in Algorithm 2.4.2.

Definition 2.4.1. An *extended finite automaton* is 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where:

1. Q is a finite set of states,
2. Σ is a finite input alphabet,
3. δ is a mapping $Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$,
4. $q_0 \in Q$ is an initial state,
5. $F \subseteq Q$ is a set of final states

Definition 2.4.2. Conversion of *EFA* to *DFA* consists of one function which for the given state determines a set of all states which are reachable by using of ϵ transition. This function is denoted by ϵ*−closure*:

$$\epsilon{-}closure(q) = \{p \mid \exists w \in \Sigma^* : (q, w) \vdash^* (p, w)\} \text{ where: } q, p \in Q$$

This function can be further generalized, so that the argument of the function can be a set $T \in Q$:

$$\epsilon{-}closure(T) = \bigcup_{s \in T} \epsilon{-}closure(s)$$

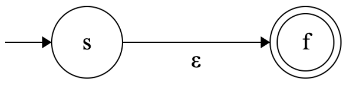
Algorithm 2.4.1. The conversion of a regular expression to EFA.

Input: A regular expression r describing the regular set R over Σ.

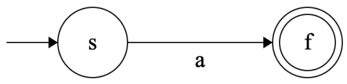
Output: An extended finite automaton M for which holds $L(M) = R$.

Method:

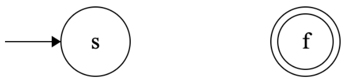
1. Decompose r into its primitive component according to recursive definition of regular sets (expressions).
2. (a) For the expression ϵ construct the automaton:



   (b) For the expression $a, a \in \Sigma$ construct the automaton:

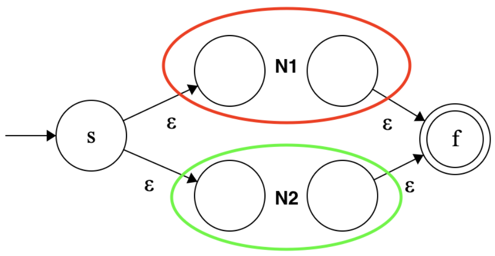


(c) For the expression $\emptyset$ construct the automaton:

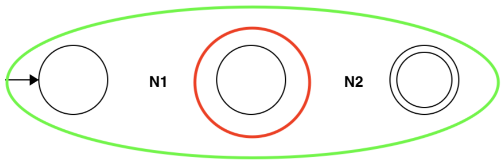


(d) Let $N1$ is the automaton accepting the language which is specified by the expression $r1$ and let $N2$ is the automaton accepting the language which is specified by expression $r2$.

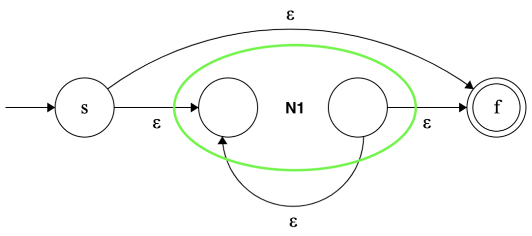
i. For the expression $r1 + r2$ construct the automaton:



ii. For the expression $r1r2$ construct the automaton:



iii. For the expression $r1^*$ construct the automaton:



Algorithm 2.4.2. The conversion of EFA to DFA.

Input: An extended finite automaton $M = \{Q, \Sigma, \delta, q_0, F\}$.

Output: A deterministic finite automaton $M^{'} = \{Q^{'}, \Sigma, \delta^{'}, q_0^{'}, F^{'}\}$

Method:

1. $Q^{'} = 2^Q \setminus \{\emptyset\}$.

2. $q_0^{'} = \epsilon{-}closure(q_0)$.
3. $\delta^{'} : Q \times \Sigma \to Q^{'} \cup \{nedef\}$ defined as:
   - Let $\forall T \in Q^{'}, a \in \Sigma : \overline{\delta}(T, a) = \bigcup_{q \in T} \delta(q, a)$.
   - Then for each $T \in Q^{'}, a \in \Sigma$:
     - if $\overline{\delta}(T, a) \neq \emptyset$ then $\delta^{'}(T, a) = \epsilon{-}closure(\overline{\delta}(T, a))$,
     - otherwise $\delta^{'}(T, a) = nedef$
4. $F^{'} = \{S \mid S \in Q^{'} \wedge S \cap F \neq \emptyset\}$.

Chapter 3

Baseline – Plogchecker

As mentioned before, *Plogchecker 2.0* is a reimplementation of *Plogchecker* [12] with several improvements. This chapter describes basic principles and algorithms used within *Plogchecker*. The chapter is organized as follows. Section 3.2 talks about the way how to specify system properties. Next, Section 3.1 describes the *Plogchecker* structure and its components. Then, Section 3.3 is dedicated to the monitoring algorithm itself. And finally, Section 3.4 summarizes the results of the *Plogchecker* benchmark from *computation time* and *memory consumption* point of view.

3.1 Data-Flow Structure

Plogchecker can be divided into four main parts, property specification, execution trace input, filter, and monitor. Property specification and execution trace input parts serve as external inputs which the end-user should specify. Filter and monitor are internal parts of the tool, and in the end, the end-user does not need to know about their internal functionality.

Property specification is a file that contains information about verified system properties and their events. Furthermore, this file can (but it is not mandatory) define parametric constraints over individual event parameters. Next, *Plogchecker* should be compatible with any data format of the input execution trace. It can process this input from a specified file (offline monitoring) or standard input (online/offline monitoring).

The *filter* is a process that selects only relevant events from the input execution trace. When the filter determines that a specific event is relevant for monitoring in the current context, the event is further passed to the monitor. The event's relevance is determined based on property specification and its defined events.

The *monitor* is the core of the monitoring algorithm. It is responsible for managing individual monitoring instances (DFA instances). In the case of parametric monitoring, it is also responsible for mapping events and their parameters to the monitoring instances. Finally, it is responsible for yielding monitoring verdicts about property violations.

3.2 Property Specification

In *Plogchecker*, the system properties are specified in a single *YAML* file. This file is then parsed, and after that, runtime monitors are created from parsed properties. One can find the grammar of the property specification file in Listing 3.1, which is defined using the

Bakus-Naur form (BNF). The grammar specifies three main parts, properties, events, and constraints. One can see the example of the property specification file in Listing 3.2.

The property file must contain at least one from properties or bad properties sections. The properties section specifies a sequence of events that the user would want to see in the execution trace of the monitored system. On the other hand, the bad properties section determines the unwanted behavior of the monitored system. Each item in these sections is defined as a tuple (`id:events`), where `id` is a unique identifier of the property and `events` define a sequence of events using ERE language.

The events section defines execution events themselves. Each event consists of the tuple (`id:event_value`), where `id` is a unique identifier of the event and `event_value` represents a specific event that should be matched with incoming events from the execution file. The `event_value` value is specified using RE.

Finally, the constraints section defines parametric constraints over individual event parameters. *Plogchecker* introduces following constraint operators `'='`, `'<'`, `'<='`, `'>'` and `'>='`, where operator `'='` is the only n-ary operator. One can specify the value of the operand as a parameter or constant. The parameter value is a reference to some event parameter. On the other hand, the constant parameter can be specified either as a number or string value.

```
<propertyfile> ::= [<propertieslist>] [<badpropertieslist>] <eventslist>
[<constraintslist>]
<propertieslist> ::= properties : \n <propertydef> {<propertydef>}
<badpropertieslist> ::= badproperties : \n <propertydef> {<propertydef>}
<propertydef> ::= <letter> : <regex> \n
<eventlist> ::= events : \n <eventdef> \n {<eventdef>}
<eventdef> ::= <letter> : <regex> \n
<constraintslist> ::= constraints : \n <constraintdef> {<constraintdef>}
<constraintdef> ::= <equal_expr> | <comparison_expr>
<equal_expr> ::= - <position_par> = <position_par> {= <position_par>}\n
<comparison_expr> ::= <position_par>|<number> <op>
    <position_par>|<number>\n
<position_par> ::= <letter> . <number>
<op> ::= '<' | '>' | '<=' | '>='
```

Listing 3.1: Grammar of the property specification file defined in BNF.

```
properties:
    file_usage: ou*c
events:
    o: open\((.+)\)
    u: use\((.+)\)
    c: close\((.+)\)
constraints:
    - o.1 = u.1 = c.1
```

Listing 3.2: Example of the property specification file.

3.3 Monitoring Algorithm

Plogchecker's monitoring algorithm consists of three main execution phases. This process is executed for every incoming relevant event atomically. That means that every such event must go through these phases. Note that the input of the monitoring algorithm is an incoming event accepted by the *filter* module. The meaning of each phase is described in the following part. Finally, the end of this section briefly explains the process of generating violation notifications for a specific property.

Phase 1

Phase 1 checks every DFA instance that is created from user-defined system properties. Suppose an automaton instance can execute the transition from starting state w.r.t. to the received event. In that case, the *monitoring algorithm* duplicates this automaton instance and executes the transition on the duplicated instance. The new instance is then added to the unfinished automaton instances (UAI) pool. Note that such input events are called *creation* events in the *runtime verification* field.

Phase 2

The second phase checks every instance from the *UAI* pool. If there exists some instance that can execute transition w.r.t. to the input event, then the transition is executed.

Phase 3

The third phase checks every configuration tree. The configuration tree is a data structure that maps parameter instances to their corresponding values. *Plogchecker* creates a unique configuration tree for every property and every unique creation event. The monitoring algorithm then traverses through every such configuration tree and checks if it can execute the transition from the current node of the tree. If yes, the monitoring algorithm creates a new DFA instance `A` that contains an events sequence derived from the root of the specific configuration tree to the current node. It then executes the transition on DFA instance `A` w.r.t. received event. Finally, DFA instance `A` is then added to the UAI pool.

Example 3.3.1 (**Configrutation tree**)**.** Consider previously defined property file for „*open, use, close*“ file example from Listing 3.2. The input execution trace could look, for example, like this (individual events divided by `'.'` character):

$$\textbf{open}(f1).\textbf{use}(f1).\textbf{close}(f1).\textbf{use}(f1).\textbf{close}(f1)$$

One can find the configuration tree for such execution trace and property file in Figure 3.1.

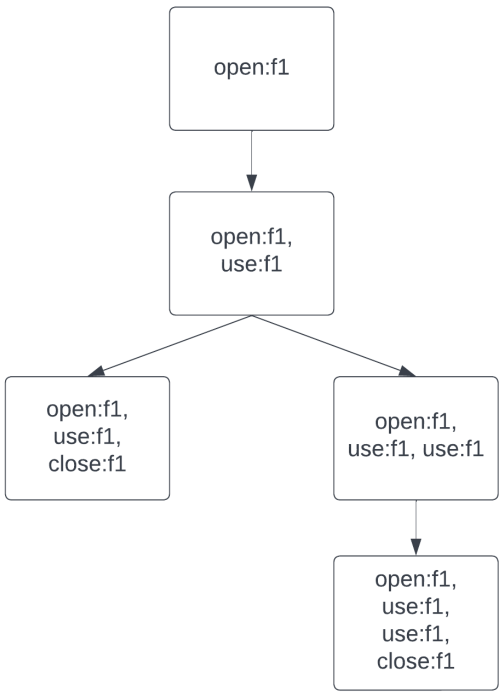


Figure 3.1: Configuration tree for execution trace defined in Example 3.3.1.

Generating Violation Notifications

After the DFA instance executes the transition, the monitoring algorithm checks if the instance has reached the final state. If yes, the instance is removed from the UAI pool. If it was the instance created from a *bad property* specification, the information about the violation of the given property is propagated to the final report. Moreover, suppose some unfinished DFA instances exist from *good property* specification at the monitoring end (no more incoming events). In that case, the information about the violation of the given property is propagated to the final report.

3.4 Plogchecker Benchmark

Input data to the benchmark process of the *Plogchecker* were generated using the *Pair-Wise Coverage* (PWC) criterion [4]. PWC criterion is defined as follows. Given a set of N input characteristics with each characteristic having L possible values, a set of test cases R is generated. Each test case from R contains N values, one from each input characteristic. Furthermore, the set of test cases R covers all possible pairs of input characteristic values. Moreover, one can observe the following input characteristics when using *Plogchecker*:

1. **Property**. One can define *parametric* or *non-parametric* properties or both at the same time.
2. **Property type**. One can specify *good* or *bad* properties or both at the same time.
3. **Number of good properties**. One can specify any number of good properties ranging from one to potentially infinity. The test set of this benchmark contains

three characteristic values, 0, 1, and >1 (more than one good property specified). Values 1 and >1 are essentially the same in the context of functionality, but one would want to see *Plogchecker* behavior if he adds more system properties to the verification process.

4. **Number of bad properties**. One can specify any number of bad properties ranging from one to potentially infinity. Characteristic values are the same as in the *good properties* case.

5. **Timeout**. All automaton instances keep track of the number of incoming events in which they are idle (no transition made). One can then specify the number of idle events required, after which the monitoring algorithm can delete the specific automaton instance.

6. **Garbage**. This input option contains two arguments. The first argument specifies the required number of open automaton instances to start deleting unused ones. The second argument specifies the number of automaton instances to be deleted in one deletion process.

7. **Input log size**. This is not an input parameter, but three input log sizes are specified for better evaluation of this benchmark: long (`X Mb`), medium (`X / 2 Mb`), short (`X / 4 Mb`). In the case of online monitoring, one could think of infinite sequences of events (infinite log size), but such input parameter would not be monitorable.

It is important to say that input characteristics 5 (Timeout) and 6 (Garbage) were omitted from the benchmark. The reason for this is simple. One cannot simply specify the best time to start garbage-collecting unused or inactive automaton instances. Based on this, the benchmark could be biased in favor of *Plogchecker* or the other way. Knowing input characteristics, one can compute a test set based on the PWC criterion. Finally, Table 3.1 summarizes all test cases available for this benchmark.

Test Case ID	**properties**	**property type**	**number of good properties**	**num of bad properties**	**log size**
Test 01	parametric	good	1	0	short
Test 02	parametric	bad	0	1	long
Test 03	parametric	combined	>1	>1	medium
Test 04	non_parametric	good	>1	0	long
Test 05	non_parametric	bad	0	>1	short
Test 06	non_parametric	combined	1	1	medium
Test 07	combined	good	1	0	medium
Test 08	combined	bad	0	1	short
Test 09	combined	combined	>1	1	short
Test 10	combined	combined	1	>1	long
Test 11	parametric	bad	0	1	medium

Table 3.1: Test cases generated using *Pair-Wise Coverage* criterion.

One can observe two main metrics while executing the benchmark on *Plogchecker*: *computation time* and *memory consumption*. Such metrics are further observed in the following paragraphs. Moreover, all benchmark executions are limited with fifteen minutes of computation time.

The benchmark was executed using the Python *memory-profiler*[1] package and its time-based memory tracking executable *mprof*. This package supports various backends for monitoring memory usage (psutil, tracemalloc, posix, and more). The default backend is set to *psutil*, which measures *resident set size* metrics. Therefore, for the needs of this benchmark, the *mprof* executable was run with default backend settings. Finally, the benchmark was performed on a 2,7 GHz Quad-Core Intel Core i7 MacBook Pro with 16 GB RAM.

Computation Time

As one can see, monitoring non-parametric properties is not a problem in terms of computation time. On the other hand, monitoring parametric properties generates some additional overhead. In this case, computation time rises depending on input file size (test cases 2 vs. 11) or the number of monitored properties (test cases 1, 2, and 11 vs. 3).

In some cases, monitoring parametric and non-parametric properties simultaneously generates the most extensive overhead on the monitored system (test cases 7 and 10). Keep in mind that, in this case, the correct input log file contains parametric and non-parametric events. The existence of non-parametric events means that automaton instances processing parametric events cannot transition to the final state as effectively as monitoring only parametric properties. This means that there are significantly more automaton instances to work with for every received event. Moreover, every received event has to iterate over all automaton instances in already mentioned execution loops.

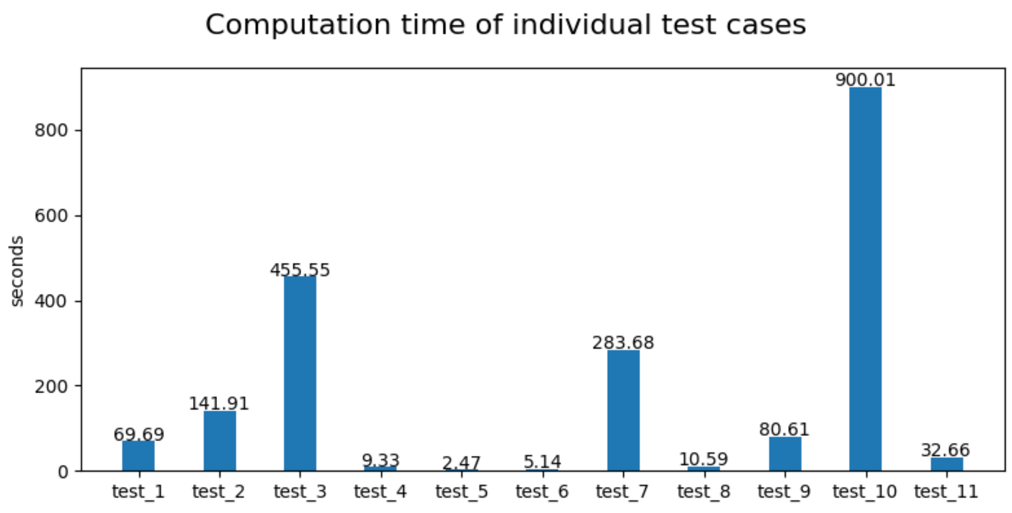


Figure 3.2: Computation time of individual test cases.

While monitoring computation time consumption, one can also be interested in the computation time of individual functions of the monitoring tool. Figure 3.2 shows the top two computation time functions. Note that the data in the following table was retrieved from test case number 2 with a total computation time equal to 141.91 seconds. This table shows data only for test case number 2 because other test cases generate a similar

[1] https://pypi.org/project/memory-profiler/

time percentage against total monitoring time. In other words, functions `read_event` and `evaulate_constraints` are in the top two positions in the context of total computation time per function in all test cases.

Function name	**Number of calls**	**Total time**	**Time per call**
read_event	30158	139.94	0.0041
evaluate_constraints	22448350	21.68	8.65096989e-7

Table 3.2: Computation time of top two functions of the *Plogchecker*.

As one can see, the monitoring algorithm spends approximately 140 seconds (98% against total computation time) of the computing time in the function `read_event`. This function essentially wraps already mentioned three main execution phases. Such observation only strengthens the hypothesis that these three phases generate the most overhead on the monitored system.

Memory Consumption

Figure 3.3 shows the maximum memory used during the whole monitoring process. As one can see, the conclusion is more or less the same as in the case of computation time. The amount of consumed memory increases either with increasing input log size or with the combination of parametric and non-parametric properties.

Figures 3.4 and 3.5 show the progress of memory consumption over time. Both figures have the same meaning. They are divided just for better visualization. The memory consumption for non-parametric properties (test cases 4, 5, and 6) has stabilized over time which is essentially expected behavior. In the case of only parametric properties, one can see that memory consumption has increasing behavior. In such test cases, the increasing behavior of memory consumption over time with new incoming parametric events is expected.

Finally, in the combined monitoring of parametric and non-parametric properties, the memory consumption over time is increasing as well. The increase behavior is not observed from the combined monitoring group, only when monitoring bad and no good properties. Such observation indicates that monitoring bad properties generates a smaller overhead than monitoring good properties. However, such behavior is expected when one assumes that the input log trace mainly contains good property traces.

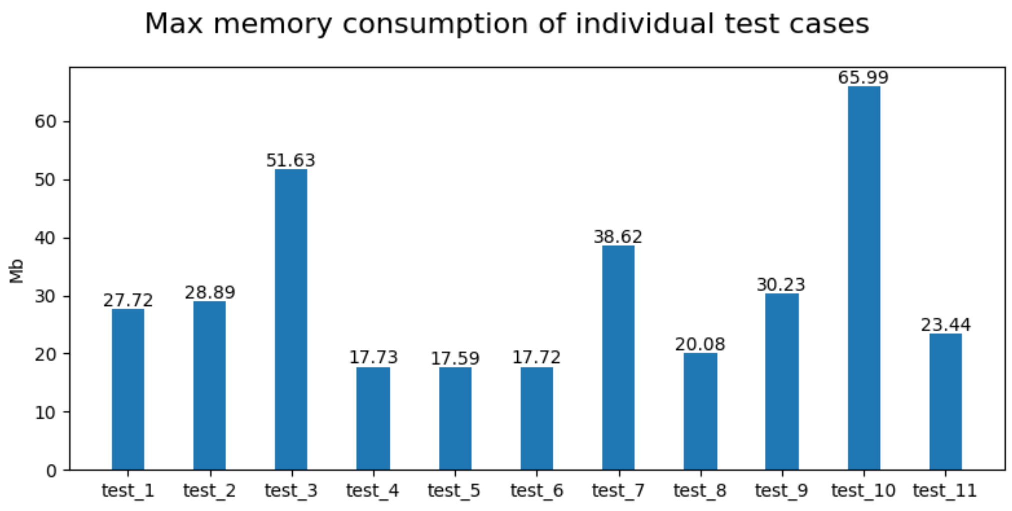


Figure 3.3: Maximum memory consumption detected in each test case.

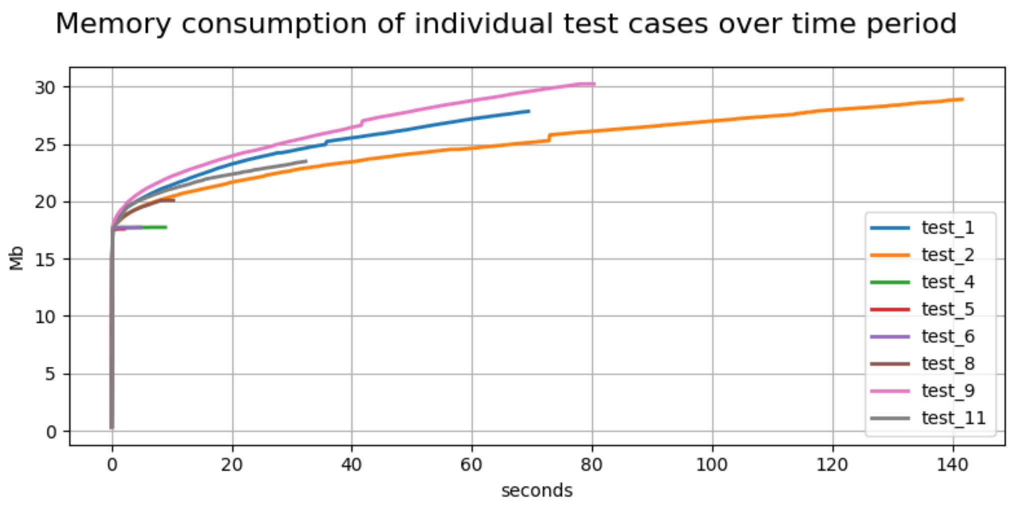


Figure 3.4: Memory consumption depending on logical time.

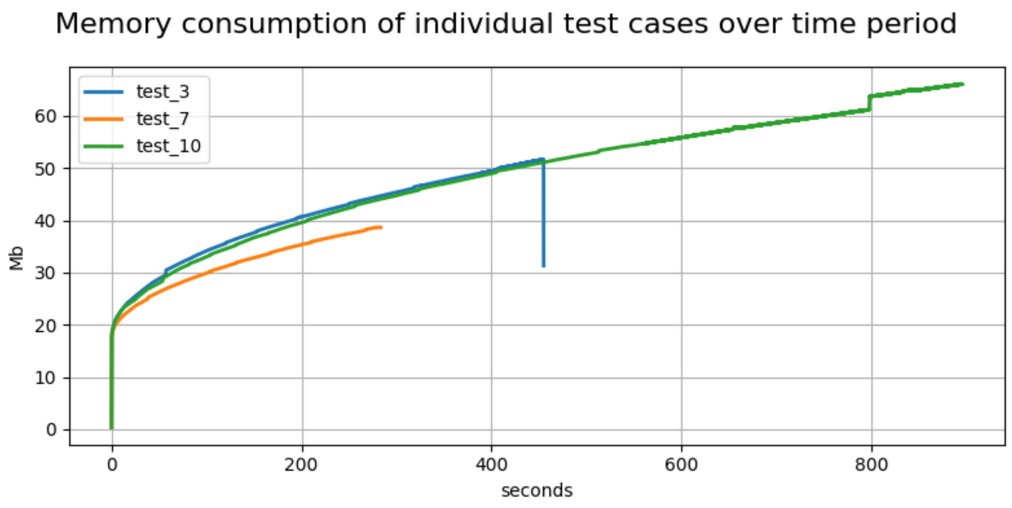


Figure 3.5: Memory consumption depending on logical time.

The benchmark showed that *Plogchecker* is an optimal *runtime verification* tool when monitoring small inputs or only one kind of property (parametric or non-parametric). However, once one starts increasing input log size or monitor parametric and non-parametric properties simultaneously, *Plogchecker* tends to increase the computation time needed or consume too much memory. Of course, increasing memory consumption or computation time is expected with increasing log inputs. Still, it would be optimal to slow down memory consumption increase from Figure 3.5 or spike in computation time from Figure 3.2.

Chapter 4

Proposal of Enhancements in Plogchecker 2.0

This chapter is dedicated to the proposed enhancements of *Plogchecker* within *Plogchecker 2.0*. As already mentioned in Section 3.4, *Plogchecker* is not optimal when monitoring parametric and non-parametric properties simultaneously or monitoring bigger size input logs. Thus, enhancement of the monitoring algorithm itself is proposed in Section 4.1. When monitoring parametric properties, the user should be able to specify data types of individual parameters. Specification of data types over individual parameters is discussed in Section 4.2. Next, Section 4.3 deals with the discard operator, which can be applied over any specific event sequence. This operator is not present in ERE language family but is important for monitoring bad properties. And finally, as mentioned in Section 3.4, *Plogchecker* offers the option to specify when to start garbage-collecting unused or inactive automaton instances. However, the end-user cannot define such parameters correctly, as this can depend on many factors (the hardware on which the monitor is running, log size, or combination of monitored properties). Thus, the garbage collecting process should be completely autonomous. This topic is discussed in Section 4.4.

4.1 Monitoring algorithm

As described in Section 3.4, *Plogchecker* generates bigger overhead when monitoring either good and bad properties simultaneously or bigger input logs. Such input to the *Plogchecker* causes almost constant spawning of new automaton instances. Those instances are then always checked against new incoming execution events. Such system behavior can potentially cause unacceptable computation times or memory consumptions. In the worst-case scenario, the monitoring algorithm can either never reach the final decision about monitored properties or crash due to out-of-memory error.

Plogchecker 2.0 uses DFA theory similar to its predecessor. However, instead of spawning new automaton instances, the processing of incoming events is slightly different. Consider DFA instance that represents some monitored property defined using RE. *Plogchecker 2.0* then represents every state of this DFA instance by some, yet undefined, table structure. Every table structure contains values corresponding to events parameters defined in the property file. These parameter values then contain references to parameter values either from other tables or the inner table. Such references between parameters values then create paths. One such path represents one DFA instance execution over specific path parameters values.

Motivation Example

Consider property file defined in Listing 3.2 and its only property. Finally, consider the following DFA instance that represents such property definition (transitions labeled as event ids):

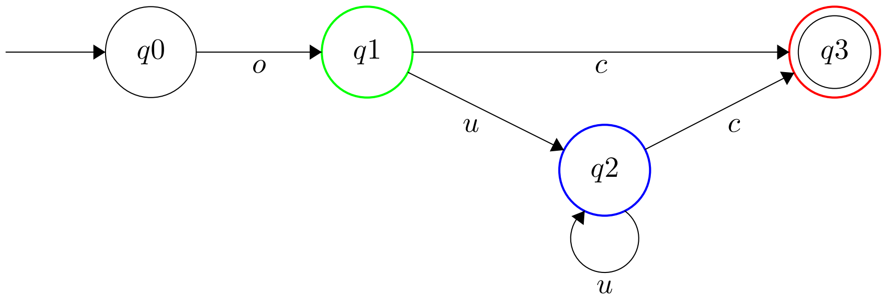


Figure 4.1: DFA template for property defined in Listing 3.2.

The execution trace for such property could look, for example, like this:

$$\textbf{open}(f1).\textbf{use}(f1).\textbf{use}(f1).\textbf{close}(f1)$$

While monitoring such trace, *Plogchecker* would spawn up to four automaton instances. One can find a visualization of the individual instantiated automaton executions (transitions labeled by incoming execution events) in Figure 4.2. Note, that the two automatons in the middle seem to be the same. But this is not the case, since we are interested in all possible monitoring sequences from the runtime verification point of view. Thus, the transition from `q1` to `q2` in the first automaton is made by the *first use* event. On the other hand, the same transition in the second automaton is made by the *second use* event.

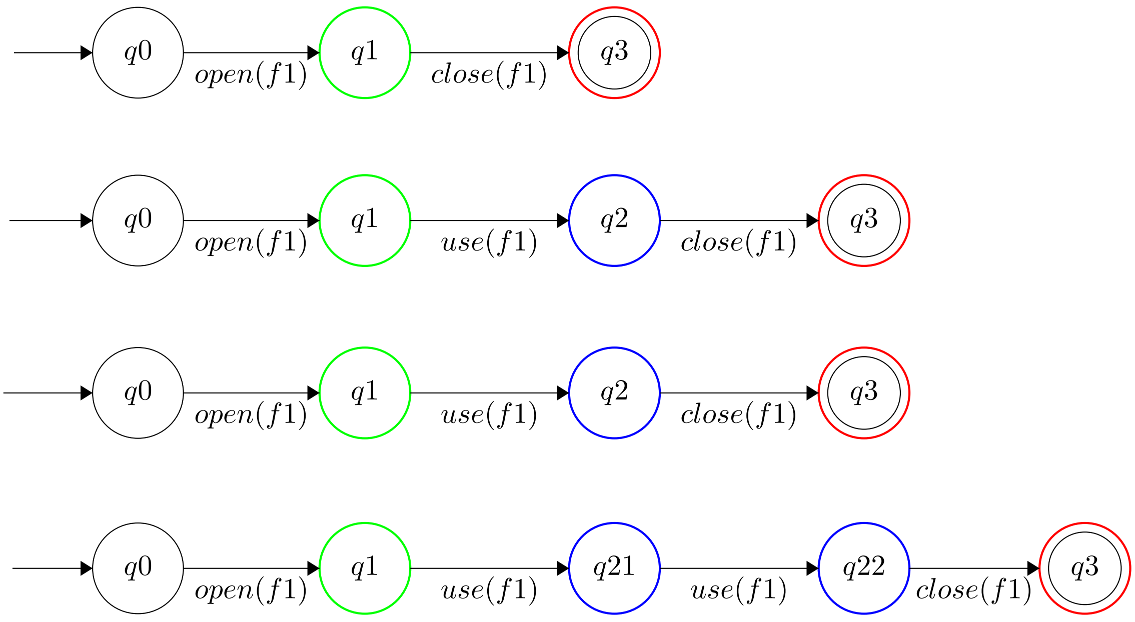


Figure 4.2: Spawned DFA executions over trace defined earlier and property from Listing 3.2.

On the other hand, *Plogchecker 2.0* would create three tables, each representing one state (except starting state) of DFA instance from Figure 4.1. Each incoming event is then added to its corresponding table (if possible). Furthermore, the added table item is then connected with items from other tables or even inner table based on some, yet undefined, rules. One can see the example of such tables in Figure 4.3, where arrows represent transitions (connections) between table items.

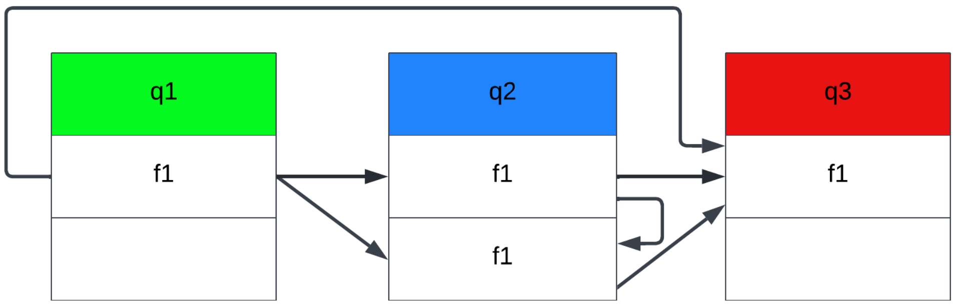


Figure 4.3: Example of *Plogcheckers 2.0* table structures for property from Listing 3.2 and top execution trace.

Suppose that transition between two table items can be represented by '→' symbol and that one item of the specific table can be represented by 3-tuple=(t,r,v) where:

- **t** is the table identifier,
- **r** is the row number of the specific item (rows indexed from 0),

- **v** is the value of the specific item.

Then, all paths from Figure 4.3 can be written as follows:

1. (q1,0,f1) $\rightarrow$ (q3,0,f1),
2. (q1,0,f1) $\rightarrow$ (q2,0,f1) $\rightarrow$ (q3,0,f1),
3. (q1,0,f1) $\rightarrow$ (q2,1,f1) $\rightarrow$ (q3,0,f1),
4. (q1,0,f1) $\rightarrow$ (q2,0,f1) $\rightarrow$ (q2,1,f1) $\rightarrow$ (q3,1,f1)

As one can see, these paths correspond precisely to the DFA instances from Figure 4.2. Moreover, the method of converting DFA instance states to table structures and representing DFA transitions by references to other table items seems to be promising in the context of *computation time* and *memory consumption* saving. Firstly, the monitoring algorithm does not have to work with potentially hundreds of spawned automaton instances at once. Secondly, it is obvious that, for example, for state *q1*, the enhanced monitoring algorithm saves the value of some specific event parameter only once. This value is then just reused when referencing other table items. On the contrary, *Plogchecker* copies the same value to all corresponding DFA instances, which significantly increases *memory consumption* and potentially *computation time.*

Preliminaries

This Section defines two main building blocks of monitoring algorithm within *Plogchecker 2.0*. Firstly, *Table* structure, that was already mentioned in the motivation example, is defined in Definition 4.1.1. And secondly, Definition 4.1.4 formally defines binary relation ϕ. Both definitions are then followed by examples.

Definition 4.1.1 (Table). Consider the definition of parametric trace and event from Definition 2.2.3. Let X_e be a set of parameters used within event $e \in \Sigma$. Furthermore, let V_e be a set of values corresponding to event's e parameters. Then, it is obvious that table structure for event e represents parameters to values mapping $e(\theta_e)$ where θ_e is a partial function in $[X_e \rightharpoonup V_e]$. Put simply, *Plogchecker 2.0* contains unique table structures for every event from defined properties. Each table contains mapping over specific event parameters domain and their corresponding values.

Example 4.1.1 (Table). Consider property file example from Figure 3.2. From this example, it is obvious that:

- $\Sigma = \{o, u, c\}$,
- $X_o = X_u = X_c = \{f\}$, parametric constraints implicate that each event expects one parameter of the same value, lets mark this parameter as f,
- $V_o = V_u = V_c = V^*$, where V can be generally any value, but lets limit this set to $V = \{f1, f2, f3\}$.

Plogchecker 2.0 will then create three tables each corresponding to one event $e \in \Sigma$. Furthemore, lets consider the following execution trace:

$$\mathbf{open}(f1).\mathbf{use}(f1).\mathbf{close}(f1).\mathbf{open}(f3).\mathbf{open}(f2).\mathbf{close}(f3)$$

Then, one can see the tables content after the trace is received by the monitoring algorithm in Figure 4.4. Table $T1$ represents mapping in $[X_o \rightharpoondown V_o]$, table $T2$ represents mapping in $[X_u \rightharpoondown V_u]$, and finally table $T3$ represents mapping in $[X_c \rightharpoondown V_c]$.

T1	T2	T3
f1	f1	f1
f3		f3
f2		

Figure 4.4: Example of *Plogchecker's 2.0* table structures for property defined in Listing 3.2.

Note, that if there would be property defined, for example, as $p1$: ou^*cu, then the events set Σ is equal to $\Sigma = \{o, u_1, c, u_2\}$ and incoming parametric event `use(a)` corresponds to both u_1 and u_2 events.

Definition 4.1.2. Let Σ_{ERE} be a set of basic events defined by *ERE*. Furthermore, let τ be a function that maps basic event $e \in \Sigma$ to its corresponding event $e_{ERE} \in \Sigma_{ERE}$.

Example 4.1.2. Consider property file example from Figure 3.2. It is obvious that:

- $\Sigma = \{o, u, c\}$
- $\Sigma_{ERE} = \{o, u^*, c\}$

The result of the function τ over every event $e \in \Sigma$ is as follows:

- $\tau(o) = o$
- $\tau(u) = u^*$
- $\tau(c) = c$

Definition 4.1.3 (Function len**).** Function len over event $e \in \Sigma$ is defined as:

$$len(e) = \begin{cases} 0 & \text{iff } \tau(e) = e^* \\ 1 & otherwise \end{cases}$$

Definition 4.1.4 (Relation ϕ**).** Suppose that for any defined parametric property, one can enumerate its events in ascending order. For example, property $p1$: abc can we rewritten as $p1$: $a_0b_1c_2$. It is then obvious that event a is expected before event b and both of the events before event c. The binary relation ϕ is then defined as:

$$\forall e_i(\theta), e_j(\theta) \in \Sigma(X) : e_i(\theta)\ \phi\ e_j(\theta) \iff j = i + 1 \lor$$
$$j > i + 1 \ \land\ \forall k:\ i < k < j \ \land\ len(e_k) = 0 \lor$$
$$j = i \ \land\ len(e_i) = 0$$

Example 4.1.3 (Relation ϕ). Consider property file example from Figure 3.2. Where:

- $\Sigma = \{o, u, c\}$,
- X is a set of events parameters,
- $\tau(o) = o,\ \tau(u) = u^*$ and $\tau(c) = c$.

Then, the relation ϕ over $\Sigma(X)$ is:

1. $o(\theta)\ \phi\ u(\theta)$,
2. $o(\theta)\ \phi\ c(\theta)$,
3. $u(\theta)\ \phi\ c(\theta)$,
4. $u(\theta)\ \phi\ u(\theta)$.

Algorithm

One can see the pseudocode of the monitoring algorithm in Algorithm 4.1. There is also pseudocode of the `traverse_paths` function in Algorithm 4.2. Function `traverse_paths` is called from Algorithm 4.1 and is directly responsible for creating transitions between individual table items.

Monitoring algorithm sequentially processes every incoming event $e \in w_p$. For each such event it iterates through each group of tables from the pTs. The following process is then same for each group of $Ts \in pTs$. For every event e there is a query for relevant table mappings (stored in *event_tables* variable) from Ts w.r.t. event e. *Relevant* table mappings w.r.t. some specific event e are such table mappings that map parameters of event e to its corresponding values. The algorithm then iterates trough every table $t_e \in$ *event_tables*. In every iteration it checks if it can add this event to the event table t_e (function `check_table`). To add some event to its corresponding table, the event must satisfy user-defined parametric constraints, which depend only on the event in question and on no other (if no parametric constraints defined it always returns true). For example, that one of its parameters should be equal to some constant (for more information about constraints, check Section 5.2).

Next, if the `check_table` function returns *true*, the `get_starting_tables` function returns the starting tables t_s w.r.t. table t_e, event e, and user-defined parametric constraints. If there are no parametric constraints defined or return value of the `get_starting_tables` function should be *NULL*, then it returns all tables that are in relation ϕ with table t_e. If t_s is not equal to *NULL*, the monitoring algorithm then iteratively picks relevant items from each table mapping $t \in t_s$ w.r.t. event e. *Relevant* items w.r.t. event e are such items that satisfy user-defined parametric constraints (if no constraints defined, relevant items are all items from t). Such items are then passed to the `traverse_paths` recursive function together with event e, t_e and t.

The function `traverse_paths` recursively traverses all paths with the root in the item $r \in ri$. Firstly, it checks if r is an item that was added in the current run of Algorithm 4.1 for a given event e (lines `2` - `18`). If yes, the function continues with the next item from ri. Such check is important, for example, in the context of a property with two identical events following each other. Otherwise, the event e and item r are connected if and only if the table mappings t_e and t_i satisfy the binary relation ϕ from Definition 4.1.4 and user-defined parametric constraints over e and r are satisfied. The function then iterates trough every reference ref to other item from r. It checks for ref table t_{ref} and recursively calls function `traverse_paths` with t_{ref}, ref, e and t_e. The function ends if it has iterated through all available references from r and its return value is whether the event e is connected to at least one table item.

After calling the `traverse_path` function for each table mapping $t \in ts$, the Algorithm 4.1 proceeds with the following last step. It adds an event e to the table t_e after the following condition is met. The given event is either *creation* or the `traverse_path` function returned true.

Algorithm 4.1: Pseudocode of the monitoring algorithm

```
Data: execution trace $w_p \in \Sigma(\theta)^*$; table mappings grouped by individual
      properties $pTs$
1  for $e \in w_p$ do
2      for $Ts \in pTs$ do
3          event_tables ← $\{t_e(\theta)\ |t_e(\theta) \in Ts\ \wedge\ t_e(\theta)$ is relevant to $e\}$
4          for $t_e \in$ event_tables do
5              if check_table($t_e$, $e$) then
6                  connected ← False
7                  creation_event_table ← is_creation_event_table($t_e$)
8                  $t_s$ ← get_starting_tables($t_e, e$))
9                  for $t \in t_s$ do
10                     ri ← get_relevant_items($t, e$)
11                     connected ← traverse_paths($t, ri, e, t_e$)
12                 end
13                 if connected $\vee$ creation_event_table then
14                     add_to_table($t_e, e$)
15                 end
16             end
17         end
18     end
19 end
```

Algorithm 4.2: Pseudocode of the *traverse_paths* function called from Algorithm 4.1

```
Data: relevant items $ri$ w.r.t. to event $e \in w_p$; corresponding table mapping $t_i$
      w.r.t. $ri$; event $e$; corresponding table mapping $t_e$ w.r.t. $e$
def traverse_paths($t_i$, $ri$, $e$, $t_e$):
    connected $\leftarrow$ False
    for $r \in ri$ do
        if $currently\_added\_item(r)$ then
            continue
        end
        $t_i \leftarrow$ get_item_table($r$)
        if $t_e \not\subset t_i \wedge contraints\_fulfilled(e, r)$ then
            connect($e, r$)
            connected $\leftarrow$ True
        end
        for $ref \in r.references$ do
            $t_{ref} \leftarrow$ get_item_table($ref$)
            connected $\leftarrow$ traverse_paths($t_{ref}, ref, e, t_e$) $\vee$ connected
        end
    end
    return $connected$
end
```

4.2 Specification of Parametric Properties

Even though *Plogchecker* provides a way to specify event parameters in their corresponding properties, there is no way to specify their data types. In recent years, there has been a development of many tools which aim to add more expresivity to their formal property specification languages. That is why *Plogchecker 2.0* tries to enrich its predecessor formal property specification language with an option to specify parameters data types.

Plogchecker 2.0 provides an option to specify parameter data types using Logstash[1] filter plugin *Grok*[2]. Put simply, *Grok* is used to parse unstructured data into structured data. Moreover, it provides a way to match some line of input file against *RE* and map specific line parts into their dedicated fields. Finally, *Grok* has a huge support from many programming languages, which makes this plugin ideal to use within *Plogchecker 2.0* input log parsing.

Basics of Grok

The basic syntax of *Grok* filter can be defined as **%{SYNTAX:SEMANTIC}** where:

- **SYNTAX** is a name of a pattern that will match the content of specific line of the input log. Each pattern is then defined by its name and corresponding *RE*.
- **SEMANTIC** is the identifier to the piece of text being matched.

[1] https://www.elastic.co/logstash/
[2] https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html

Furthemore, *Grok* provides huge variety of basic patterns, ranging from *networking* (e.g., IPv4 or IPv6), *dates* (years, months, and days), *paths* (e.g. unix path or windows path), to some very basic patterns such as *numbers*, *strings* or *usernames*. However, if one would not find suitable pattern for his needs he can always create the custom one in the already specified syntax.

Example 4.2.1 (**Usage of Grok patterns in Plogchecker 2.0**)**.** Lets consider already discussed „*open, use, close*“ file property example. The property file describing such property from Listing 3.2 could be rewrited using *Grok* pattern *WORD*. This pattern is defined by `\bw+\b` regular expression. One can see the rewrited property file in Listing 4.1. This property file indicates that monitoring algorithm expects three kind of events. Each event contains one parameter of type *WORD* and name *p1*.

```
properties:
    file_usage: ou*c
events:
    o: open\(%{WORD:p1}\)
    u: use\(%{WORD:p1}\)
    c: close\(%{WORD:p1}\)
constraints:
    - o.p1 = u.p1 = c.p1
```

Listing 4.1: Example of property file for „*open, use and close*“ file example using Grok patterns.

As you can see, the language for specifying properties is more expressive using *Grok* patterns. One can define parameter data types and constraints over them pretty easily, which makes the *Grok* plugin ideal to use within *Plogchecker 2.0*. However, when specifying constraints over parameters, there is one issue which is discussed in the following Section.

Lexical Analysis Over Event Parameter Constraints

As stated in Section 3.2, *Plogchecker* provides a functionality to specify constraints over defined event parameters. In the framework of *Plogchecker 2.0* this functionality should be extended by *lexical analysis* over these constraints (on top of support for parameter data types). The analysis would ensure that the specified constraint over the parameters is correct from the lexical point of view. For example, that the user does not try to compare a number and a string with each other. Or that he does not try to apply an operator to a parameter of a data type that does not support this operator.

4.3 Discarding Monitored Sequences

As mentioned in Section 3.2, the specification of properties can be divided into so-called good and bad properties. However, the discard operator (`'!'`) is strictly applicable only to bad properties. Therefore, this section will further deal with this operator only from the bad properties point of view.

In Section 3.2, it was mentioned that a bad property is basically a sequence of events that the user would not want to see in the verified system log. This behavior is correct from the runtime verification point of view but not sufficient. The user should be able to specify that the monitoring of a bad property is only of interest until the arrival of a certain sequence of events. If such a sequence is received, monitoring of the bad property is discarded. If not, and the bad property is met, then only at this point would the user be notified of the bad property being met.

Example 4.3.1 (Usage of discard operator). Consider the following bad property p:

p: abc

The property will be fulfilled when three consecutive events, a, b and c, are received by the monitoring algorithm. Further, suppose the user would like to express that after receiving event d between events b and c, the given property p can no longer be fulfilled. Then, the property p would be defined as follows:

p: abd!c

4.4 Garbage Collecting

Garbage collection is a very important runtime verification process. Without it, the amount of consumed memory would grow directly proportional to the incoming events (precisely, events that pass through the filter object). Therefore, the following garbage collection method is proposed within *Plogchecker 2.0*.

Let's suppose that the table objects is a critical section area. Next, let two threads run simultaneously. One thread is the monitoring algorithm itself and the other thread would represent the garbage collection process. These would synchronize with each other using a simple global mutex lock.

The thread for the monitoring algorithm would try to acquire this global lock with each incoming event. If it fails, it will be suspended until it succeeds in acquiring this lock. After processing the incoming event, it would release this lock and the whole process would repeat with the next incoming event.

The thread representing garbage collecting would try to get this lock every `X` seconds. The X-second sleep interval is an acceptable solution, since there is no need to try to garbage collect during the entire run of the monitoring algorithm. After getting the lock, the thread would decide whether it makes sense to garbage collect in the following way. It would look at the tables belonging to a specific property and if the size of at least one table exceeds the set treshold `Z` then it would try to remove all *closed event sequences* from these tables. A closed event sequence is an event sequence that satisfies a given property and that is not further elaborated. Removing all closed event sequences during monitoring is probably the simplest and most correct garbage collection technique from a runtime verification point of view.

A practical example of which sequences could be removed and which could not be removed can be found in Example 4.4.1. Moreover, one can find the flow diagram on execution of such two threads in Figure 4.6.

Example 4.4.1 (Garbage collection of closed event sequences). Imagine the following property `p` defined in ERE language:

```
p: ab(c*d)*
```

Next, suppose that following is the input trace (events divided by `'.'`):

`a.b.d.c.a.b`

And finally, the populated table mappings (from Section 4.1) would look like this (omit arrows as references, instead, created event sequences are colour highlighted):

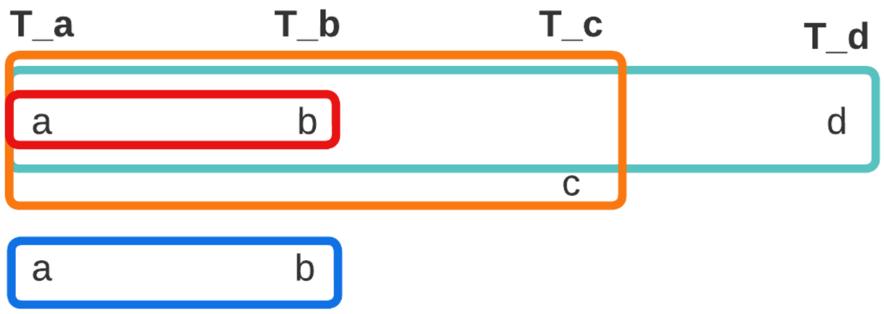


Figure 4.5: Contents of table structures after receiving input trace **a.b.d.c.a.b**.

As one can see in Figure 4.5, there are four event sequences created:

- **sequence0** (red colour): a b
- **sequence1** (green colour): a b d
- **sequence2** (orange colour): a b c
- **sequence3** (blue colour): a b

From these sequences only *sequence0*, *sequence1* and *sequence3* are closed, and will be removed by garbage collecting process. Finally, *sequence2* is waiting for event `d` to arrive, so it is considered unfinished and in progress and will not be removed by the garbage collection process. Such a proposed garbage collection process guarantees the detection of all *minimal* sequences that meet the bad property specification. For example, *sequence3* satisfies property `p`, but once it is garbage collected, the monitoring algorithm will no longer detect other sequences with potential incoming events `c` and `d`.

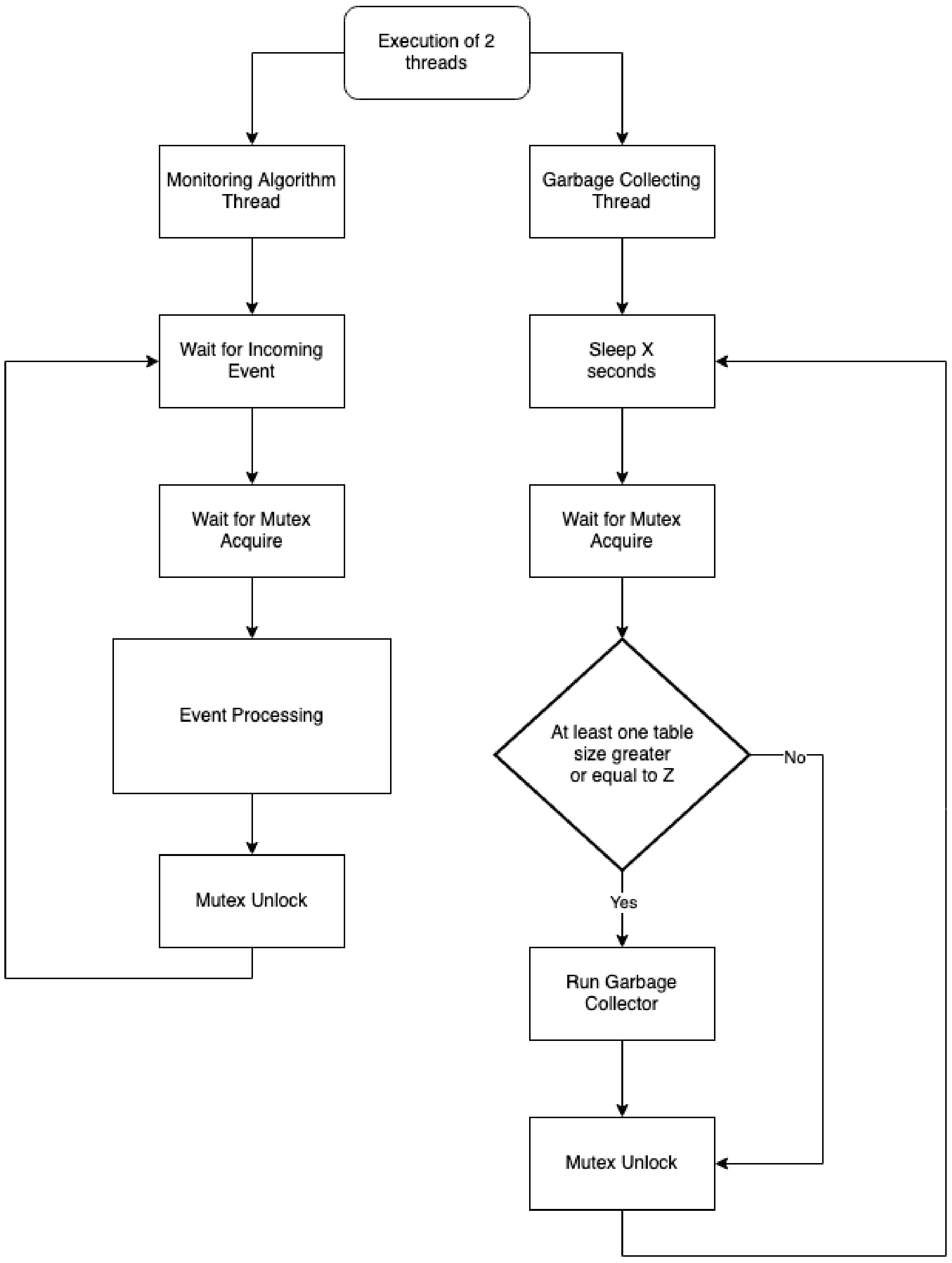


Figure 4.6: Monitoring algorithm and Garbage Collection flow diagram.

4.5 Implementation Language

The original version of *Plogchecker* is implemented in the *Python* programming language. However, Python is generally known for being slow. This is caused, for example, by the fact that it is an interpreted programming language or that it lacks a Just in time compiler unlike other interpreted languages (e.g. Java or .NET), or that it uses the so-called Global interpreter lock [15]. Based on that, Python is not suitable for applications where the speed in terms of computation time or the efficiency of memory usage is important.

Therefore, for the needs of *Plogchecker 2.0*, the *Go* programming language was chosen. Go is a statically typed, compiled programming language designed by Google [14]. It promises development speed similar to languages like Python with performance similar to languages like C++ or C. Although Go is not portable because it compiles source code into platform-specific binary code, it allows cross-platform compilation for different operating systems and architectures. The cross-platform compilation is managed by two environment variables, *GOOS* and *GOARCH*. GOOS stands for target operating system, and GOARCH stands for the target architecture. One can see the example of Go cross-platform compilation for different operating systems (Windows, Linux, Darwin) and architectures (amd64, arm, arm64) in Listing 4.2.

```
# Windows operating system and amd64 architecture
env GOOS=windows GOARCH=amd64 go build -o hello.exe .

# Linux operating system and arm architecture
env GOOS=linux GOARCH=arm go build -o hello .

# Darwin operating system and arm64 architecture
env GOOS=darwin GOARCH=arm64 go build -o hello .
```

Listing 4.2: Example of Go cross-platform compilation for different operating systems and architectures.

Chapter 5

Implementation Details of Plogchecker 2.0

This chapter describes the implementation details of *Plogchecker 2.0*. Section 5.1 describes the properties file. It is precisely how it is parsed, what data types it supports within the event parameters, and what rules it must follow. Finally, it explains the minor changes within the properties and bad properties sections.

Section 5.2 deals specifically with the constraints section of the property file since this section have seen the most changes in terms of property specification. It describes other data types that are supported over those supported in the properties section. Furthermore, it describes an implementation of lexical analysis over the specified constraints.

Next, Section 5.3 deals with the table structure and its items. Then, building such tables from the specified properties is explained in more detail. Finally, the monitoring process, from data-flow point of view is briefly explained in Section 5.4. The section is then mainly focused on the monitoring process output.

5.1 Property File

This section describes one of the essential parts of the *Plogchecker 2.0*, without which monitoring would be impossible. The individual parts are discussed from the parsing process of the property file, what rules the property file has to meet, and finally, the supported data types within the events section.

Parsing Process

In *Plogchecker 2.0*, the structure for property definition remains the same as in *Plogchecker* (Listing 3.1). This structure is only enriched with support for some additional operators and parameter data types discussed below. Moreover, the property definition preserves the syntax of the *YAML* files.

Parsing of such files with defined properties is controlled by two structures, *PropertiesConfRaw* and *PropertiesConf*. The *PropertiesConfRaw* structure serves as an initial mapper of the content of the specified *YAML* file into a structured form. The properties, badproperties and events sections must be specified as key-value, where both are expected as string data type. The constraints section is defined as an array of string expressions.

After *Plogchecker 2.0* parses the constraints section to the internal form of the tool (more on this topic in section 5.2), the *PropertiesConf* structure is initialized with all values from the properties, badproperties and events sections. This structure is then supplemented with the parsed data from the constraints section. The functionality described above is taken care of by the *property_file.go* and *constraints.go* files.

Parameter Data Types

As mentioned in the enhancement design, *Plogchecker 2.0* supports specifying event parameter data types using the Logstash filter plugin *Grok*. Therefore, it supports the following data types within the parameter specification:

1. **NUMBER**
   - Supports any kind of positive or negative numbers in *decimal system*.
   - **example values**: 14, -42
2. **WORD**
   - Supports any kind of strings that satisfy `\bw+\b` regular expression.
   - **example values**: string, string_42
3. **DATESTAMP_RFC1123**
   - Supports date string in *RFC1123* format.
   - **example value**: Mon, 14 May 1998 10:12:00 UTC
4. **DATE_ISO8601**
   - Supports date string in *ISO8601* format.
   - **example value**: 1998-05-14 10:12:00

Rules

The property definition file defines several rules that must be met for it to be considered valid. These rules are listed and explained below.

Rule 1

Rule 1 defines that the event identifier must begin with a capital letter and may or may not continue with any word character equivalent to $[a-z, A-Z, 0-9, \_]$. In other words, event identifier must satisfy the following definition specified by a regular expression: `^[A-Z](\w)*$`. One can see an example of a violation of such a rule in Listing 5.1.

```
properties:
    p1: a
bad_properties:
events:
    a: "a" # event 'a' does not start with capital letter
constraints:
```

Listing 5.1: Example of a violation of Rule 1.

Rule 2

Rule 2 defines that a property file must specify at least one good or bad property. One can see an example of a violation of such a rule in Listing 5.2.

```
# No property defined within properties or bad_properties section
properties:
bad_properties:
events:
    A: "a"
constraints:
```

Listing 5.2: Example of a violation of Rule 2.

Rule 3

Rule 3 defines that every event used in the properties or bad_properties section must be defined in the events section. One can see an example of a violation of such a rule in Listing 5.3.

```
properties:
    A B C # event 'C' is not defined
events:
    A: "a"
    B: "b"
constraints:
```

Listing 5.3: Example of a violation of Rule 3.

Rule 4

Rule 4 defines that the discard operator can only be used in the bad_properties section.

Rule 5

```
properties:
    A B
events:
    A: 'a %{WORD:p1}'
    B: 'b %{WORD:p1}'
constraints:
    - "A.p1 = B.p2" # event 'B' does not contain parameter with name 'p2'
    - "A.p1 = C.p1" # event 'C' is not defined
```

Listing 5.4: Example of a violation of Rule 5.

Rule 5 states that every event and every event parameter used within the constraints section must be defined in the events section. One can see an example of a violation of such a rule in Listing 5.4.

Properties and BadProperties Sections

The specification of the properties remained the same as in the *Plogchecker*. There are, however, a few exceptions worth highlighting. As already mentioned, properties (event sequences) are specified using extended regular expressions.

Because events can be specified as a word and not just as a single character, *Plogchecker 2.0* implements the *concatenation* operator as a space. This change has been made both for greater user-friendliness and to make it easy to tell where one event ends and another begins.

Another modification is adding support for the *discard* (`'!'`) operator, which is introduced in Section 4.3. All other operators from the extended regular expression family are preserved. Finally, to put it all together, *Plogchecker 2.0* supports the following operators within properties sections:

```
        ' ' - concatenation, '|' - alternation, '*' - iteration
   '+' - positive iteration, '{}' - bounded iteration, '!' - discard
```

Of the closed iteration expressions, only two are supported out of all available, namely, the expression `'{n}'` (given expression has to be present exactly n times) and `{n,m}` (given expression has to be present from n to m times). Finally, every specified property needs to satisfy defined grammar from Listing 5.5.

```
regex = branch [ { | branch } ]
branch = { expr }
expr = ( regex ) | expr operator | EVENT
operator = '*' | '+' | '{' n '}' | '{' n1 , n2 '}' | '!' | ϵ
```

Listing 5.5: A grammar expressing properties definitions in extended Backus-Naur form, enriched with ! operator, partially adopted from [12].

5.2 Constraints Section

This section describes the supported operators within the constraints property file section. And finally, it explains the process of *lexical analysis* over individual constraints.

Constraints Data Types

In addition to the data types outlined in the previous chapter, the constraints section supports two more. Namely, the `DURATION` and `BOOL` data types. From the specification point of view, the `DURATION` data type is a string that must satisfy the following regular expression `\d+h\d+m\d+s`. Intuitively, it is a value in which it is possible to specify the number of hours, minutes, and seconds. A possible example of its specific value is, for example, `1h15m0s`. Such a value is further parsed into the internal runtime representation as a `Duration` data type which is part of the Go time package[1].

The `BOOL` data type logically defines only two values, true and false. These values are specified as strings and reserved as two unique values for lexical analysis and constraints parsing.

[1] https://pkg.go.dev/time

These data types are viewed as *constant* types. This means that they are not defined by any parameter but their data type is automatically detected by their value. *Plogchecker 2.0* also supports the definition of `WORD` and `NUMBER` data type constants within constraints section. Thus, for example, the value `'42'` is viewed as a constant of the `NUMBER` data type. And finally, the value `'abcd'` is viewed as a constant of the `WORD` data type.

Supported operators

Plogchecker 2.0 supports the following operators within the constraint specification:

1. `'='` - equal,
2. `'!='` - not equal,
3. `'>'` - greater,
4. `'<'` - less,
5. `'>='` - greater or equal,
6. `'<='` - less or equal
7. `'+'` - plus,
8. `'-'` - minus,

All operators are binary operators. Input operands can be of type *parameter* or *constant*. Moreover, there are two special functions over *WORD* data type supported:

1. `is_substr`(s1, s2) - returns whether *s1* contains sub-string *s2*.
   - where *s1* is a *parameter* of the *WORD* data type and *s2* is a *parameter* or *constant* of the *WORD* data type
   - **return value data type**: *BOOL*
2. `length`(s1) - returns length of *s1*.
   - where *s1* is a *parameter* of the *WORD* data type
   - **return value data type**: *NUMBER*

Lexical Analysis

Lexical analysis over individual constraints is implemented using the lexical analysis framework *lexmachine*[2]. *Lexmachine* divides the input string into sub-strings (lexemes) and categories (tokens) based on specified patterns. These patterns are specified using regular expressions.

One can register the new pattern and associated token using the `Add` function over the `Lexer` structure defined by this framework. Moreover, there is the possibility of a kind of dynamicity where one does not have to specify the corresponding token but the function that calculates this token. When defining several patterns simultaneously, previously unmatched text can be matched to more than one pattern. *Lexmachine* solves this situation by selecting a pattern defined earlier in the pattern hierarchy.

[2] https://github.com/timtadh/lexmachine

Plogchecker 2.0 uses the terms `Literal` and `Token` to distinguish between key values and values that can be dynamic within the constraints section. Literals include operators, and tokens include data types, both of which were specified in Section 4.1. With all the information defined above, Listing 5.6 shows the pseudo-code for the registering patterns and their associated tokens within *Plogchecker 2.0*.

The result of the tokenization process over a specific constraint is an array of individual tokens and literals (ordered by their position in a specific constraint). *Plogchecker 2.0* then concatenates this array into a single string using the empty string separator. Then, it checks if this string is available in the `AvailableLexConfigurations` structure. If yes, the result of lexical analysis over specified constraint is that the constraint is valid. Otherwise, the message about the probable error is propagated to the user. One can find a full list of such configurations in Appendix A.

Example 5.2.1 (**Lexical analysis**)**.** Consider a simple property file from Listing 5.7 and its one specified constraint `A.p1 = B.p1`. There are two events with one parameter each. For both events, the parameter carries values of the *WORD* data type. Therefore, the result of the tokenization process would be `[WORD, =, WORD]`. Such an array would then be concatenated into a single string value `WORD=WORD`. Moreover, this string would be validated against the available configurations from the `AvailableLexConfigurations` structure. The result would be that from the point of view of lexical analysis this particular constraint is valid because the `AvailableLexConfigurations` structure contains a configuration string `WORD=WORD`.

```
lexer = lexmachine.NewLexer()
for _, lit := range Literals {
    lexer.Add(lit, lit)
}

lexer.Add(`\w+\.\w+`, computeTokenType()} // Compute parameter data type
lexer.Add(`length\(\w+\.\w+\)`, NUMBER)
lexer.Add(`is_substr\(\w+\.\w+, \w+\)|substr\(\w+\.\w+, \w+.\w+\)`, BOOL)
lexer.Add(`\d+h\d+m\d+s`, DURATION)
lexer.Add(`-?\d+`, NUMBER)
lexer.Add(`true|false`, BOOL)
lexer.Add(`\w+`, WORD)
```

Listing 5.6: Pseudo-code of patterns and their tokens registration.

```
properties:
 p1: AB
events:
 A: 'a %{WORD:p1}'
 B: 'b %{WORD:p1}'
constraints:
 - 'A.p1 = B.p1'
```

Listing 5.7: Simple property file example.

5.3 Creating Tables from Defined Properties

This section first presents the *Table* and *TableItem* structures in the form of a class diagram. Then, their relationships and attributes are discussed in more detail. And finally, the section is devoted to the actual building of tables from the specified properties.

Table Structure

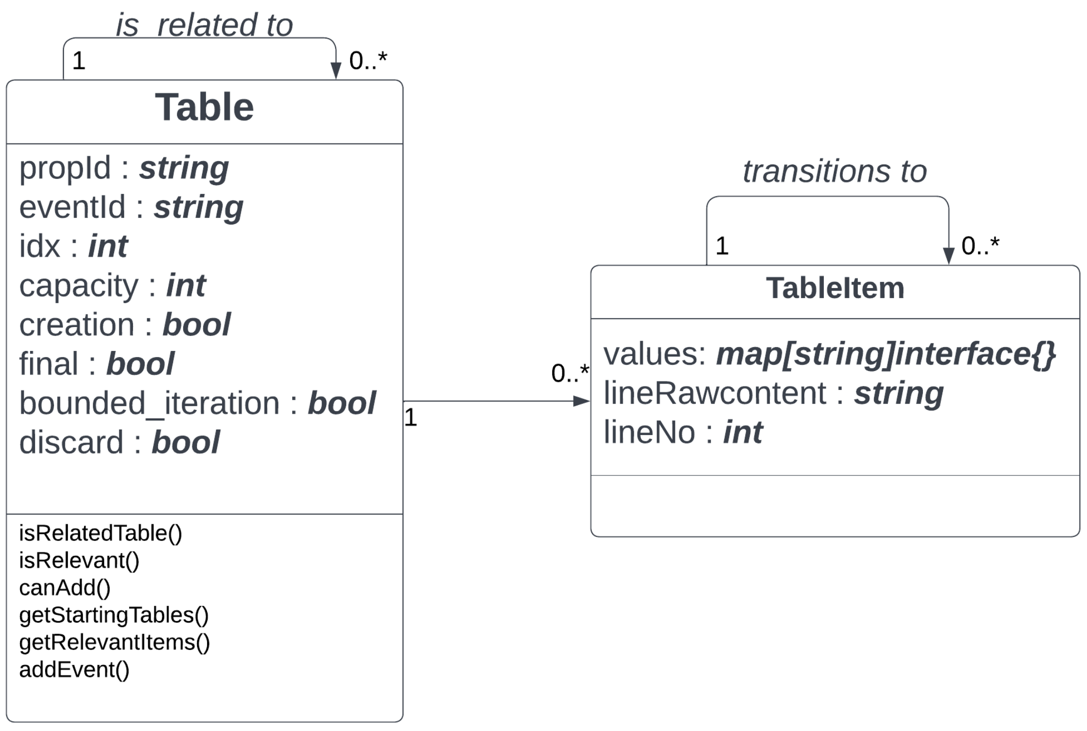


Figure 5.1: *Table* and *TableItem* structures and their relationships to each other shown by a class diagram.

As one can see in Figure 5.1, one table can be *related* from zero to many tables. This association models ϕ relations from Definition 4.1.4. For each table, *Plogchecker 2.0* implements this association as an array of pointers to other tables in the `relatedTables` variable. Moreover, one table can contain from zero to many table items. This is implemented as an array of instantiated table items in `items` variable for each table.

And finally, from each table item, there can be from zero to many transitions to another table items. Such transitions are implemented as a key-value structure for each table item `p`. The *key* is a reference to some table `T`, and the *value* is an array of references to table items of the table `T` that form a transition with item `p`. The rest of this section briefly discusses the individual attributes of both structures.

Table Attributes

- `propId` - Indicator under which property the given table belongs.

- `eventId` - Indicator which event the table processes.
- `idx` - Numeric representation of the table order within the specified property.
- `capacity` - Indicator of whether the table processes an event with positive, negative or no iteration. Possible values are as follows: `StarCap` (`'*'` operator), `PlusCap` (`'+'` operator) and `oneEventCap` (all other operators).
- `creation` - Indicator of whether the given table processes an event that *opens* a monitoring sequence within the defined property. For example, for property `'A B C'`, event `A` opens the monitoring sequence.
- `final` - Indicator of whether the given table processes an event that *closes* a monitoring sequence within the defined property. For example, for property `'A B C'`, event `C` closes the monitoring sequence.
- `discard` - Indicator whether the table processes an event that is part of an discard events sequence.

TableItem Attributes

- `values` - Key-value evaluation of the parameters of a given event. The key is the parameter identifier and the value can be, in general, a value of any data type.
- `lineRawContent` - A raw, unprocessed line from input log.
- `lineNo` - The line number from the input log on which the given event appeared.

Tables Builder

Building tables from user-defined properties starts in the `buildParameterTables` function. This function iterates through all defined properties and creates an abstract syntax tree (AST) from each property, using the Go package *goyacc*[3]. *Plogchecker 2.0* represents a given AST in the following format: `AST = [op, AST1]`, where `op` is a specific operator from the *ERE* language family (including the `'!'` operator) and `AST1` is an array whose items are either individual events or other ASTs. One can see an example of AST representation for a specific property in Example 5.3.1.

Example 5.3.1 (AST representation)**.** Consider the following property `p`:

```
p: A B* C
```

The corresponding AST for such property would look like this (note that Plogchecker 2.0 represents concatenation operator as `' '`):

```
AST = [' ', [A, [* [B]], C]]
```

Next, from such AST, it creates the corresponding array of tables and the dependencies (ϕ relations) between them. The starting point for this process is the `buildFromAst` function. Based on the defined operator, this function decides how to create tables belonging to the given operator and to the AST over which this operator works. The rest of this section describes these different ways of creating tables for a particular type of operator. Note that the tables from the examples in the individual sections below have no meaning on their own without the context of the monitoring algorithm from Section 4.1.

[3]https://pkg.go.dev/golang.org/x/tools/cmd/goyacc

Preliminaries

Consider global integer variable `tableIdx`, which is reset to zero with every new property iteration in `buildParameterTables`.

Consider two functions `getAllCreationTables` and `getAllFinalTables`, which return such tables from the *Table* structure array with the *creation* or *final* attribute set to true.

And finally, consider two functions `invalidateCreationState` and `invalidateFinalState`, that set the *creation* or *final* attribute to false for each table on the input.

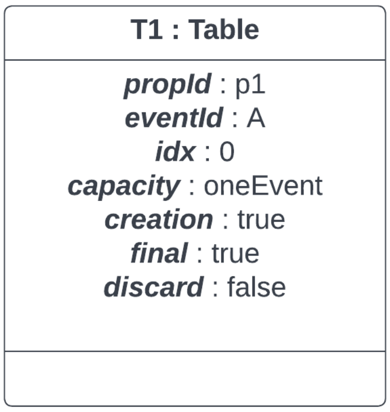
One Event – createOneEvent

Function `createOneEvent` creates and returns a table structure for the corresponding property and event with the *creation* and *final* attributes set to true. An implementation of this function is given in Listing 5.8. Furthermore, one can see an example of the created table by this function and an equivalent DFA instance for the specific property in Example 5.3.2.

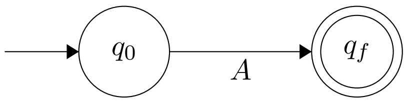
```go
func createOneEvent(propID string, event string) *Table {
    table := &Table{
        propId:           propID,
        eventId:          event,
        idx:              tableIdx,
        creation:         true,
        final:            true,
        capacity:         OneEventCap,
        discard:          false
    }
    tableIdx++
    return table
}
```

Listing 5.8: Implementation of the createOneEvent function.

Example 5.3.2. Consider the following property `p1`, `p1: A`. Then, one can see an example of created table structure `T1` and DFA instance for property `p` in Figure 5.2.



(a) *Table* structure object diagram for property *p1*.



(b) DFA instance for property *p1*.

Figure 5.2: A figure with two subfigures

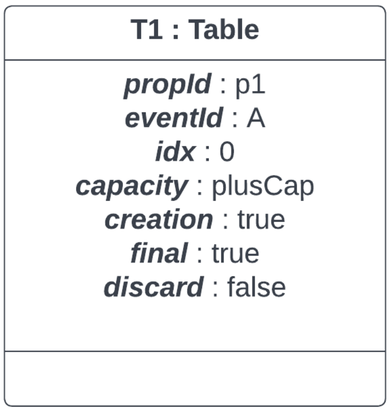
Positive and Negative Iteration – createIteration

Function `createIteration` builds an array of *Table* structures over specific operator (`'*'` or `'+'`). A simplified implementation of this function is given in Listing 5.9. At first, it calls the `buildFromAst` function, which creates an array of tables for the specified AST to which the given operator is applied. Then it checks all tables that have the *creation* (`cTs` variable) or *final* (`fTs` variable) attribute set to true. And finally, for all `cTs` tables, it creates relations (ϕ relation from Definition 4.1.4) with all `fTs` tables. Such relations create an iteration in the frame of the given AST. Within the absolute last action of this function, the capacity attribute of all tables is set to the capacity of the given operator (*starCap* for `'*'` and *plusCap* for `'+'`). Examples of this function result and equivalent DFA instance for the specific property are given in Examples 5.3.3, 5.3.4, and 5.3.5.

```
func createIteration(propID string, ast []interface{}, capacity int)
    []*Table {
    tables := buildFromAst(propID, ast)
    cTs := getAllCreationTables(tables)
    fTs := getAllFinalTables(tables)
    for _, t := range cTs {
        t.relatedTables = append(t.relatedTables, fTs...)
    }
    for _, t := range tables { t.capacity = capacity }
    return tables
}
```

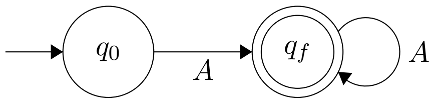
Listing 5.9: Simplified implementation of the createIteration function.

Example 5.3.3 (**Positive iteration over one event**)**.** Consider the following property `p1`, `p1: A+`. One can see an example of created table structure `T1` and DFA instance for property `p` in Figure 5.3.



T1 : Table

propId : p1
eventId : A
idx : 0
capacity : plusCap
creation : true
final : true
discard : false

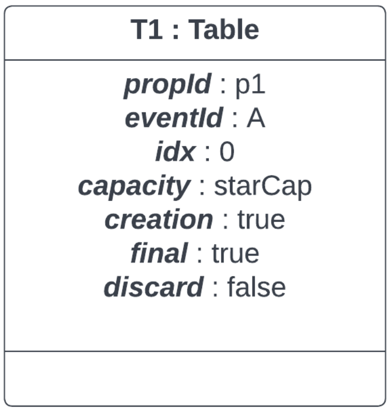


(b) DFA instance for property *p1*.

(a) *Table* structure object diagram for property *p1*. Where, T ϕ T.

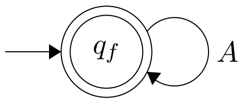
Figure 5.3: A figure with two subfigures

Example 5.3.4 (**Negative iteration over one event**)**.** Consider the following property `p1`, `p1: A*`. One can see an example of created table structure `T1` and DFA instance for property `p` in Figure 5.4.



T1 : Table

propId : p1
eventId : A
idx : 0
capacity : starCap
creation : true
final : true
discard : false



(b) DFA instance for property *p1*.

(a) *Table* structure object diagram for property *p1*. Where, T ϕ T.

Figure 5.4: A figure with two subfigures

Example 5.3.5 (**Negative iteration over events sequence**)**.** Consider the following property `p1`, `p1: (A B)*`. An example of created table structures and DFA instance for property `p` is given in Figures 5.5 and 5.6, respectively.

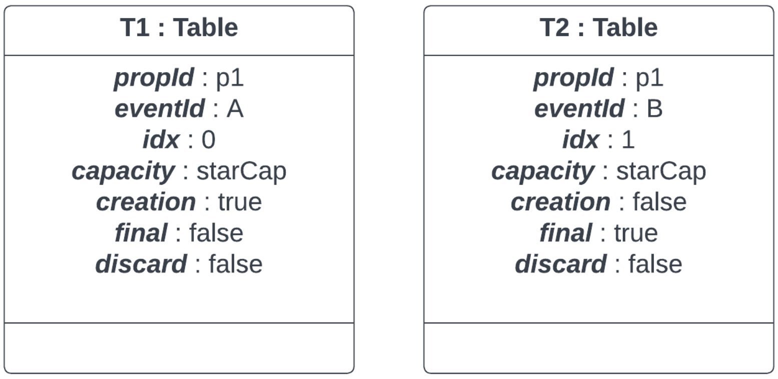


Figure 5.5: *Table* structure object diagram for property *p1*. Where, T1 ϕ T2 and T2 ϕ T1.

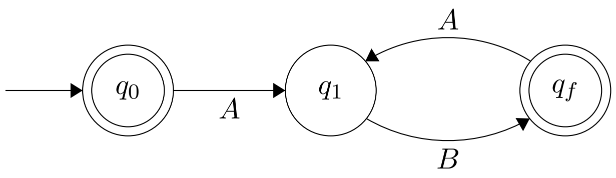


Figure 5.6: DFA instance for property *p1*.

Bounded Iteration – createBoundedIteration

The `createBoundedIteration` function creates an array of tables, and ϕ relations between them, from the given AST for which its length is equal to the `max` value. Note that specification of the `min` value is optional, and if not specified, its value is equal to `-1`. A simplified implementation of this function is given in Listing 5.10.

In each iteration, the `buildFrommAst` function is called to build an array of `newTs` tables for the given AST. Next, all tables that have the creation (`cTs` variable) and final (`fTs` variable) attributes set to true are selected from `newTs`. Then, suppose the current iteration is not equal to `1` (first iteration). In that case, all creation attributes of the `cTs` tables are set to false (the tables do not process events that open the monitoring sequence for the given AST). Moreover, for all such tables, a dependency is created with all tables from `prevFts` (variable explained later). This dependency creates a transition between the tables of the current iteration and the tables of the preceding iteration.

In the last logical step, the function decides whether to invalidate (override *final* attribute to false) all `fTs` tables. It does this if one of the following conditions is met:

- `min != -1 && i < min`: Variable `min` is specified, and the current iteration i is smaller. This ensures that the monitoring sequence will not be satisfied until the minimum number of sequences of the given AST is reached.

- `min == -1 && i != max`: Variable `min` is not specified, and the current iteration `i` has not reached `max` value. This ensures that the monitoring sequence will be satisfied only after the required number of sequences of the given AST has been reached.

And as the very last step, the `lastFts` variable is populated with all `fTs` tables and all `newTs` tables are appended to the array of result tables `res`.

```go
func createBoundedIteration(propID string, min int, max int, ast
    []interface{}) []*Table {
    var res []*Table
    var prevFts []*Table

    for i := 1; i <= maxIter; i++ {
        newTs := buildFromAst(propID, ast)
        cTs := getAllCreationTables(newTs)
        fTs := getAllFinalTables(newTs)

        if i != 1 {
            invalidateCreationState(cTs)
            for _, t := range cTs {
                t.relatedTables = append(t.relatedTables, prevFts...)
            }
        }

        if (minIter != -1 && i < minIter) ||
           (minIter == -1 && i != maxIter) {
                invalidateFinalState(fTs)
        }
        prevFts = fTs
        res = append(res, newTs...)
    }
    return res
}
```

Listing 5.10: Simplified implementation of createBoundedIteration function.

Example 5.3.6 (Exact bounded iteration). Consider the following property `p1, p1: A{2}`. One can see an example of created table structures and DFA instance for the property `p` in Figures 5.7 and 5.8, respectively.

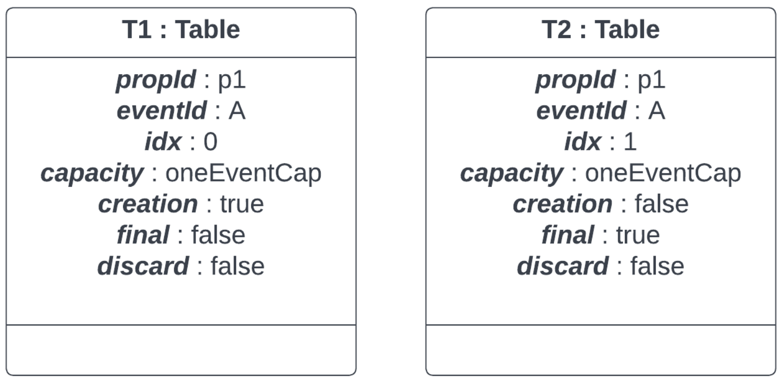


Figure 5.7: *Table* structure object diagram for property *p1*. Where, T1 ϕ T2.

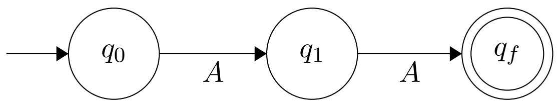


Figure 5.8: DFA instance for property *p1*.

Example 5.3.7 (**Interval bounded iteration**)**.** Consider the following property `p1, p1: A{1,2}`. One can see an example of created table structures and DFA instance for the property `p` in Figures 5.9 and 5.10, respectively.

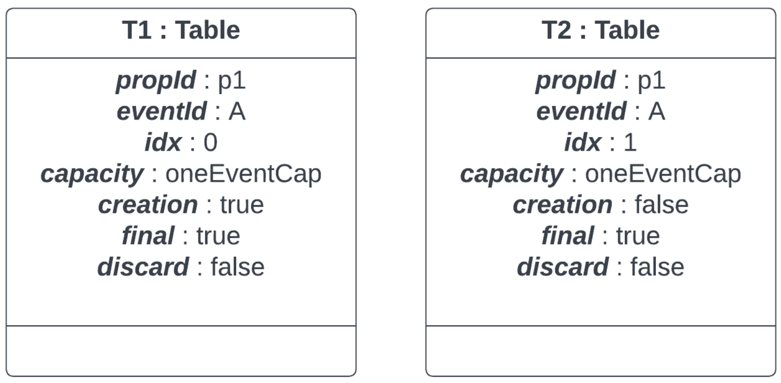


Figure 5.9: *Table* structure object diagram for property *p1*. Where, T1 ϕ T2.

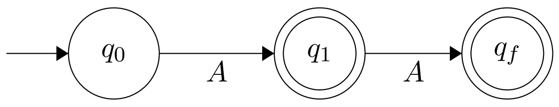


Figure 5.10: DFA instance for property *p1*.

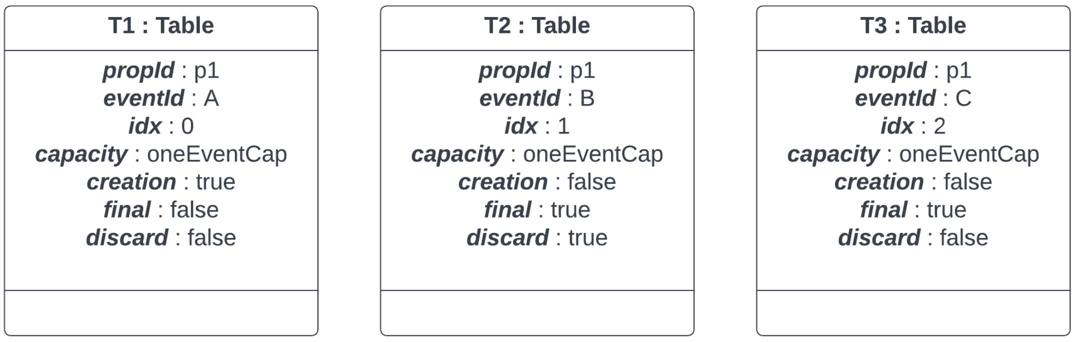
Discard – createDiscard

Function `createDiscard` builds result tables from a given AST with the `buildFromAst` function. It then iterates over every such table and sets its *discard* attribute to true. One can see a result of this function and DFA instance for the specific property in Example 5.3.8. A simplified implementation of this function is given in Listing 5.11.

```go
func createDiscard(propID string, ast []interface{}) []*Table {
    tables := buildFromAst(propID, ast)
    for _, t := range tables { t.discard = true }
    return tables
}
```

Listing 5.11: Simplified implementation of createDiscard function.

Example 5.3.8 (**Discard operator inside concatenated event sequence**). Note, that it is recommended to check the section on building tables for the *concatenation* operator before studying this example. Consider the following property `p1`, `p1: A B! C`. The corresponding table structures and DFA instance for this property can be seen in Figures 5.11 and 5.12, respectively.



T1 : Table
propId : p1
eventId : A
idx : 0
capacity : oneEventCap
creation : true
final : false
discard : false

T2 : Table
propId : p1
eventId : B
idx : 1
capacity : oneEventCap
creation : false
final : true
discard : true

T3 : Table
propId : p1
eventId : C
idx : 2
capacity : oneEventCap
creation : false
final : true
discard : false

Figure 5.11: *Table* structure object diagram for property *p1*. Where, T1 ϕ T2 and T1 ϕ T3.

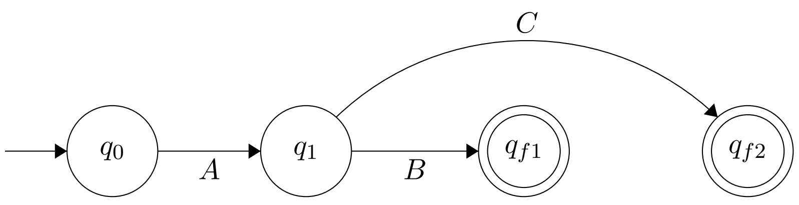


Figure 5.12: DFA instance for property *p1*.

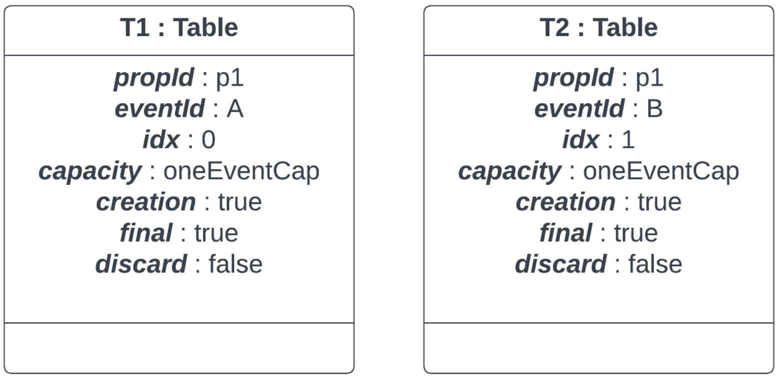
Alternation – createAlternation

The `createAlternation` function just iterates over the individual AST items over which the alternation operator is applied. For each item, the `buildFromAst` function is called, and its result is appended to the result table array `res`. The result of the `createAlternation` function and DFA instance for the specific property can be seen in Examples 5.3.9 and 5.3.10. A simplified implementation of this function is given in Listing 5.12.

```go
func createAlternation(propID string, ast []interface{}) []*Table {
    var res []*Table
    for _, part := range ast {
        tables := buildFromAst(propID, part)
        res = append(res, tables...)
    }
    return res
}
```

Listing 5.12: Simplified implementation of createAlternation function.

Example 5.3.9 (**Simple alternation of two events**)**.** Consider the following property `p1`, `p1: A|B`. The corresponding table structures and DFA instance for this property can be seen in Figures 5.13 and 5.14, respectively.



T1 : Table

propId : p1
eventId : A
idx : 0
capacity : oneEventCap
creation : true
final : true
discard : false

T2 : Table

propId : p1
eventId : B
idx : 1
capacity : oneEventCap
creation : true
final : true
discard : false

Figure 5.13: *Table* structure object diagram for property *p1*.

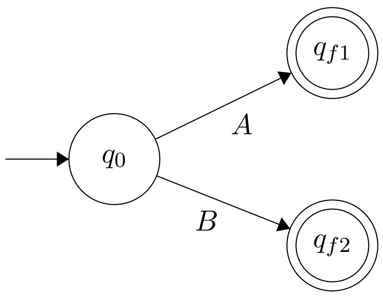
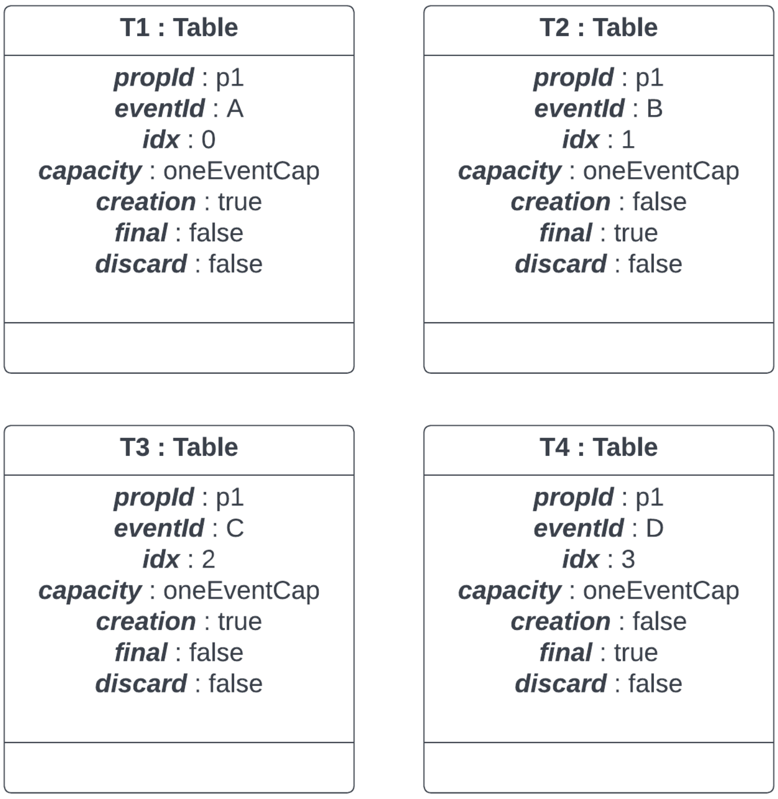


Figure 5.14: DFA instance for property *p1*.

Example 5.3.10 (Alternation of two event sequences). Consider the following property `p1`, `p1: (A B)|(C D)`. The corresponding table structures and DFA instance for this property can be seen in Figures 5.15 and 5.16, respectively.



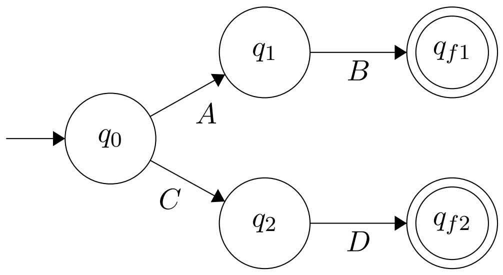
T1 : Table
propId : p1
eventId : A
idx : 0
capacity : oneEventCap
creation : true
final : false
discard : false

T2 : Table
propId : p1
eventId : B
idx : 1
capacity : oneEventCap
creation : false
final : true
discard : false

T3 : Table
propId : p1
eventId : C
idx : 2
capacity : oneEventCap
creation : true
final : false
discard : false

T4 : Table
propId : p1
eventId : D
idx : 3
capacity : oneEventCap
creation : false
final : true
discard : false

Figure 5.15: *Table* structure object diagram for property *p1*. Where T1 ϕ T2 and T3 ϕ T4.



$$\begin{tikzcd} & & q_1 \arrow[r, "B"'] & q_{f1} \\ {} \arrow[r] & q_0 \arrow[ur, "A"'] \arrow[dr, "C"'] & & \\ & & q_2 \arrow[r, "D"'] & q_{f2} \end{tikzcd}$$

Figure 5.16: DFA instance for property *p1*.

Concatenation – createConcatenation

Function `createConcatenation` iterates over the individual parts of the input AST and stores the table structures corresponding to the specific part in the result array `res`. A simplified implementation of this function is given in Listing 5.13.

```go
func createConcatenation(propID string, ast []interface{}) []*Table {
    var resTs []*Table
    for idx, part := range ast {
        newTs := buildFromAst(propID, part)
        if idx == 0 {
            resTs = append(resTs, newTs...)
            continue
        }
        cTs := getAllCreationTables(newTs)
        fTs := getAllFinalTables(resTs)
        for _, t := range cTs {
            t.relatedTables = append(t.relatedTables, fTs...)
        }
        if newTs[0].capacity != StarCap && !newTs[0].discard {
            invalidateFinalState(resTs)
        }
        var lastTable := resTs[len(resTs)-1]
        if !(lastTable.capacity == StarCap && lastTable.creation) {
            invalidateCreationState(newTs)
        }
        resTs = append(resTs, newTs...)
    }
    return resTs
}
```

Listing 5.13: Simplified implementation of createConcatenation function.

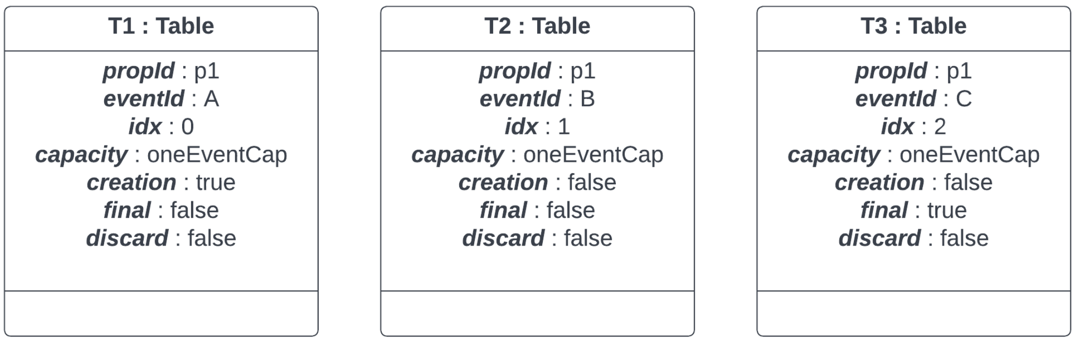
For each AST part, the `buildFromAst` function is called to create an array of tables corresponding to that part. Next, there are two ways in which the algorithm of this function can go. Firstly, if a given part is the first in the AST, the tables created for it are appended to the `res` and the algorithm continues with the next part.

Secondly, if a given part is not the first in the AST, the algorithm continues in the following way. It gets all tables from the `newTs` array with the creation attribute set to true (`cTs` variable). Next, it checks for all tables from a `resTs` array with the *final* attribute set to true (`fTs` variable). Then it creates relations for every table from `cTs` with all `fTs` tables. In other words, the sequence of events that would be accepted by the current `resTs` tables must continue in one of the tables from `cTs`. The algorithm then proceeds in two steps.

Suppose the first table from `newTs` does not have its capacity set to *StarCap* and discard attribute set to true. In that case, all final attributes of the `resTs` tables are overwritten with the value false. This means that the monitoring events sequence is not *closed* in one of the `resTs` tables but one of the `newTs` tables, with the *final* attribute set to true.

Further, if the last table from `resTs` does not have the *creation* attribute set to *true* and the *capacity* attribute set to *StarCap*, all creation attributes from `newTs` are overwritten to the value false. Invalidating all creation attributes from the `newTs` tables under the above conditions ensures that the monitoring sequence is *opened* only in one of the `resTs` tables. Finally, all `newTs` tables are appended to the `resTs` tables array. One can see an example result of this function and DFA instance for the specific property in Examples 5.3.11, 5.3.12, and 5.3.13.

Example 5.3.11 (Simple concatenation of three events). Consider the following property `p1, p1: A B C`. The corresponding table structures and DFA instance for this property can be seen in Figures 5.17 and 5.18, respectively.

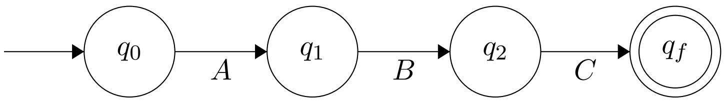


T1 : Table
propId : p1
eventId : A
idx : 0
capacity : oneEventCap
creation : true
final : false
discard : false

T2 : Table
propId : p1
eventId : B
idx : 1
capacity : oneEventCap
creation : false
final : false
discard : false

T3 : Table
propId : p1
eventId : C
idx : 2
capacity : oneEventCap
creation : false
final : true
discard : false

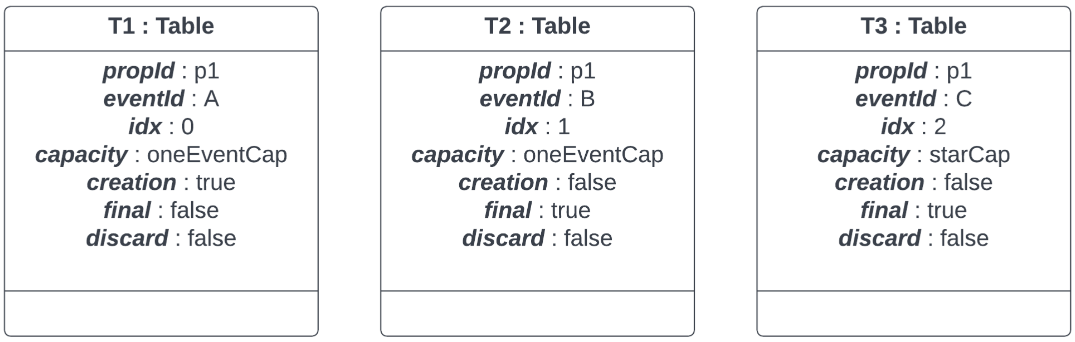
Figure 5.17: *Table* structure object diagram for property *p1*. Where, T1 ϕ T2 and T2 ϕ T3.



$$\begin{tikzcd} {} \arrow[r] & q_0 \arrow[r, "A"'] & q_1 \arrow[r, "B"'] & q_2 \arrow[r, "C"'] & q_f \end{tikzcd}$$

Figure 5.18: DFA instance for property *p1*.

Example 5.3.12 (Concatenation with negative iteration at the end). Consider the following property `p1, p1: A B C*`. The corresponding table structures and DFA instance for this property can be seen in Figures 5.19 and 5.20, respectively.



T1 : Table
propId : p1
eventId : A
idx : 0
capacity : oneEventCap
creation : true
final : false
discard : false

T2 : Table
propId : p1
eventId : B
idx : 1
capacity : oneEventCap
creation : false
final : true
discard : false

T3 : Table
propId : p1
eventId : C
idx : 2
capacity : starCap
creation : false
final : true
discard : false

Figure 5.19: *Table* structure object diagram for property *p1*. Where, T1 ϕ T2, T2 ϕ T3 and T3 ϕ T3.

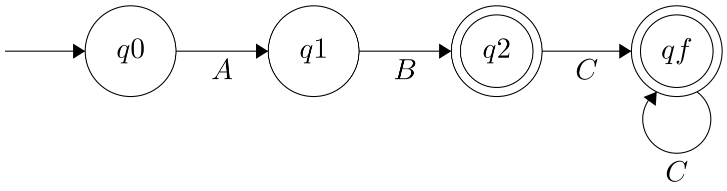
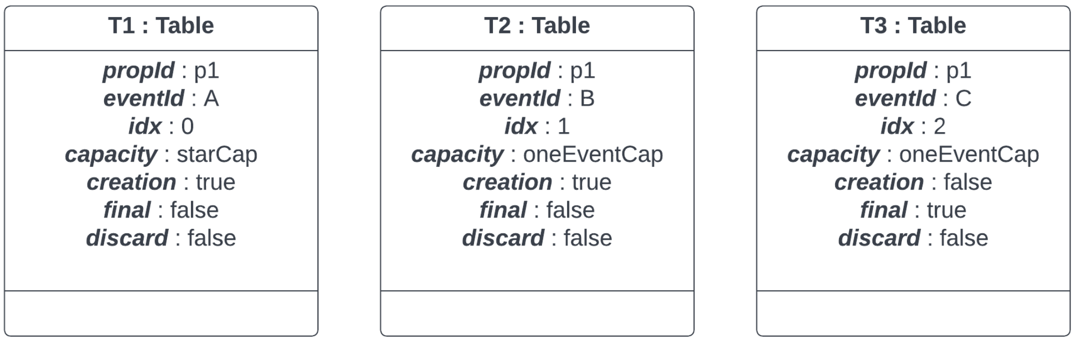


Figure 5.20: DFA instance for property *p1*.

Example 5.3.13 (Concatenation with negative iteration at the beginning). Consider the following property `p1`, `p1: A* B C`. The corresponding table structures and DFA instance for this property can be seen in Figures 5.21 and 5.22, respectively.



T1 : Table
propId : p1
eventId : A
idx : 0
capacity : starCap
creation : true
final : false
discard : false

T2 : Table
propId : p1
eventId : B
idx : 1
capacity : oneEventCap
creation : true
final : false
discard : false

T3 : Table
propId : p1
eventId : C
idx : 2
capacity : oneEventCap
creation : false
final : true
discard : false

Figure 5.21: *Table* structure object diagram for property *p1*. Where, T1 ϕ T1, T1 ϕ T2 and T2 ϕ T3.

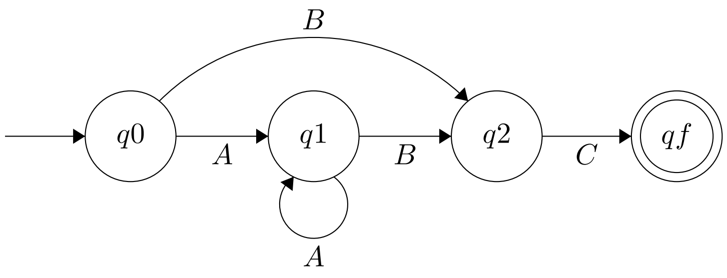


Figure 5.22: DFA instance for property *p1*.

5.4 Monitoring Process

Plogchecker 2.0 implements the whole monitoring process from the data-flow point of view in basically the same way as *Plogchecker* (discussed in Section 3.1). There are some exceptions, which are mainly related to the proposed enhancements. The most significant changes can be seen in the monitoring algorithm, which is responsible for processing incoming events. Such algorithm is implemented in the *monitor.go* and *parameter_table.go* files.

Monitor.go implements the core of the monitoring algorithm and is essentially a rewrite of Algorithm 4.1 into the Go programming language. *Parameter_table.go* implements all functions that operate over the *Table* structure (and its *TableItem* items) and are called from the core of the monitoring algorithm.

It is important to note that this algorithm runs iteratively with each incoming event for each defined property. Consider the property definition from Listing 5.7. With each incoming event that passes through the filter object, the monitoring algorithm is triggered only once because there is only one property defined. The rest of this section is dedicated to the monitoring algorithm output. More precisely, its structure and how and when it is generated.

Monitoring Output

The monitoring output (or the final report) has the task of sharing which properties have been fulfilled or violated with the user. After the end of the monitoring process, one can find the final report (`report.json` file) in the same folder from which the *Plogchecker 2.0* was executed (if not specified otherwise, see user manual in Appendix D).

Good Properties

A *good property* is satisfied if all sequences created during monitoring belonging to the given property end up in some table item from table with the final flag set to true. If some sequence does not meet this condition, but the monitoring process is not yet complete, the following sequence of incoming events can extend the sequence to meet the good property. Therefore, it is impossible to evaluate the good property sequence other than after the reception of all incoming events.

The search for such sequences starts sequentially in all tables belonging to the given property, with the *creation* flag set to true. From these tables, the algorithm then tries to search all possible sequences that were created during the monitoring. If a sequence violates a condition defined above, it is added as a good property violation to the final report.

Bad Properties

When monitoring *bad properties*, a property is violated if a monitoring sequence ends in a table item from the table with the *final* flag set to true. If such a sequence exists, it can be immediately declared violated and then added to the final report.

Imagine that, during the monitoring process, the incoming event is added as a table item to some table. If the table has the *final* flag set to true and at the same time does not operate over the event from the *discard* events sequence, the algorithm has just detected a violation of the bad property. Note that if the monitoring sequence ends in the table with the discard attribute set to true, the sequence cannot be declared a violation of the bad property (explained in Section 4.3). The monitoring algorithm then finds all tables (belonging to the given property) that have the *creation* flag set to true. And starting from these tables, it tries to find all possible sequences that end in the currently added table item. Such sequences are then added to the final report as a bad property violation.

Chapter 6

Experiments and Evaluation

This chapter is devoted to evaluating *Plogchecker 2.0* from different points of view. Section 6.1 compares the performance results between *Plogchecker* and *Plogchecker 2.0*. More precisely, it discusses two metrics, similar to Section 3.4, *computation time* and *memory consumption*. Furthermore, unit and integration tests are described in Section 6.2. And finally, Section 6.3 briefly lists the different architectures on which *Plogchecker 2.0* has been tested.

6.1 Benchmark

One of the indicators of the quality of the *Plogchecker 2.0* is how powerful it is compared to the *Plogchecker*. With data (properties and input logs) as in Section 3.4, this was calculated for two metrics; *computation time* and *memory consumption*. The benchmark was performed on the same machine as in Section 3.4. Furthermore, the computation time was measured using the system utility *time*[1], and it was limited to fifteen minutes for each test case. Within the *Plogchecker* tool, the amount of consumed memory was measured using the Python package *memory-profiler*[2]. And finally, the same, but in the context of *Plogchecker 2.0*, was done using the Go package *gopsutil*[3] and its *process* module. One can find all benchmark test cases and an automatic script for running the benchmark in the `benchmarks/` directory.

Computation Time

Table 6.1 shows the benchmark results between *Plogchecker* and *Plogchecker 2.0* in terms of computation time within individual test cases. It also highlights the speed up of *Plogchecker 2.0* against *Plogchecker* that was calculated by the following formula:

```
speed up = plogchecker_time / plogchecker2.0_time
```

As one can see, *Plogchecker 2.0* performs better in every single test case in terms of computation time. Next, it achieved an average acceleration of approximately 14x. Most notable are the cases where Plogchecker generated the highest overhead during monitoring. These are test cases 3, 7, and 10. In two of these cases, the speedup is around 30x.

[1]https://man7.org/linux/man-pages/man1/time.1.html
[2]https://pypi.org/project/memory-profiler/
[3]https://github.com/shirou/gopsutil/

Another interesting observation can be seen, for example, in test cases 2 and 3. As mentioned in Section, *Plogchecker* generates more computation time in test case 3 than in test case 2 depending on the addition of monitoring properties. It seems that *Plogchecker 2.0* responds much better to the addition of monitoring properties since its computation times are more or less equal in these two test cases.

Test ID	**Plogchecker time in seconds**	**Plogchecker 2.0 time in seconds**	**Speed up of Plogchecker 2.0**
Test 01	69.69	3.84	18.14x
Test 02	141.91	16.14	8.79x
Test 03	455.55	15.48	29.43x
Test 04	9.33	2.02	4.61x
Test 05	2.47	0.55	4.49x
Test 06	5.14	1.05	4.89x
Test 07	283.68	8.75	32.42x
Test 08	10.59	1.38	7.67x
Test 09	80.61	5.59	14.42x
Test 10	900.01	45.28	19.88x
Test 11	32.66	3.19	10.24x
Average	181.06	9.39	14.01x

Table 6.1: Measured computation time of *Plogchecker* and *Plogchecker 2.0* within individual test cases.

Memory Usage

Table 6.2 shows the memory usage of both tools during the benchmark run. The *max* column represents the maximum peak detected during a given test case run. On the other hand, the *average* column represents the average value of memory usage during a given test case run. As can be seen, *Plogchecker 2.0* performs better in terms of memory usage in every single test case. In addition, it was able to save, on average, almost twice as much memory as *Plogchecker* (1.82x).

Further, in Appendix B, it is possible to see the amount of consumed memory depending on the time. Each test case is represented by its associated figure. The most interesting test cases are the ones where the use of the automatic garbage collecting process, explained in Section 4.4, can be seen. From these figures, it can be concluded that *Plogchecker 2.0* has managed to slow down the increase of consumed memory, either in terms of adding monitoring properties or increasing the input log volume.

Test ID	**Plogchecker consumed memory in MB**		**Plogchecker 2.0 consumed memory in MB**		**Saving**	
	max	***average***	***max***	***average***	***max***	***average***
Test 01	27.72	24.38	18.00	13.83	1.54x	1.76x
Test 02	28.89	25.62	23.00	18.04	1.26x	1.41x
Test 03	51.63	40.14	23.00	19.34	2.24x	2.07x
Test 04	17.73	17.27	10.00	9.31	1.77x	1.85x
Test 05	17.59	14.78	9.00	7.20	1.95x	2.05x
Test 06	17.72	16.66	10.00	8.78	1.77x	1.90x
Test 07	38.62	31.53	20.00	17.05	1.93x	1.85x
Test 08	20.08	18.71	11.00	10.00	1.83x	1.87x
Test 09	30.23	26.35	21.00	16.34	1.44x	1.61x
Test 10	65.99	49.33	32.00	25.22	2.06x	1.96x
Test 11	23.44	21.08	15.00	12.31	1.56x	1.71x
Average	30.88	25.99	17.45	14.31	1.76x	1.82x

Table 6.2: Measured memory consumption of *Plogchecker* and *Plogchecker 2.0* within individual test cases.

6.2 Functionality Testing

Verification of the correctness of the functionality was performed using *unit* tests during implementation. Furthermore, *integration* tests were designed to verify the tool's functionality as a whole. And finally, an *exemplary* case was created, which demonstrates the real use of *Plogchecker 2.0* in practice.

Unit Tests

Unit testing was done using the *Go* package *testing*[4], in which each test function must be in the form `func TestXxx(*testing.T)`, where `Xxx` does not start with a lowercase letter of the alphabet. All unit tests are implemented in the project root directory. They can be identified by the suffix `_test.go`. These tests can be run, from the project root directory, using the `go test . -v` command where `'.'` is the current directory and `-v` enables verbose output after the command execution. The unit tests were used to test the essential elements of the tool, in particular:

- **constraints_lexer_test.go** - Correct functioning of lexical analysis over parametric constraints.
- **constraints_test.go** - Correct parsing and evaluation of parametric constraints.
- **parameter_table_builder_test.go** - Correct building of table structures from defined properties.
- **properties_file_test.go** - Correct parsing of a property files.

[4]https://pkg.go.dev/testing#section-directories

Integration Tests

Integration tests were performed using the command line utility *jd*[5]. This utility can be used to compare and flatten JSON values. The integration tests themselves are defined in the *integration_tests* folder. This folder contains all test case folders, where each folder contains:

- **properties.yaml**: properties to be monitored,
- **trace.log**: input log trace to be monitored,
- **expected_output.json**: expected final report value in JSON format.

Each test case is identified by its serial number and is divided into three different parts; parametric, non_parametric, and combined. The *parametric* test case is dedicated only to parametric events, and the *non_parametric* test case is just the opposite. Finally, the combined test case is dedicated to the *combination* of parametric and non_parametric events. Each test case is executed using the compiled *Plogchecker 2.0* tool binary with an associated log and property file, where the final report will be stored in the `report.json` file. This file is then checked against the expected result JSON value associated with the given test case (stored in `expected_output.json` file). If the two files are not equal in terms of JSON value equivalence, the test case is considered failed. The described process is automated by a bash script, which is located in the project root directory file, `run_integration_tests.sh`.

Demonstration example

The demonstration example shows the simplicity of using *Plogchecker 2.0* to monitor unexpected access to the `/etc/shadow` file. The monitoring was performed over the output of the system call, *strace*. The property file for this use case is shown in Figure 6.1. The property file specifies that the sequence to be monitored starts with an *open* or *open_at* event over the `/etc/shadow` filename. The event then returns some file descriptor in the form of a numeric character. The result of this event is a non-zero number of bytes read over the specified file descriptor.

```
bad_properties:
  shadow: "O R"
events:
  O: 'open(at)?\("/etc/shadow",.* = %{NUMBER:p2}$'
  R: 'read\(%{NUMBER:p1},.* = %{NUMBER:p2}$'
constraints:
  - "O.p2 >= 0"
  - "O.p2 = R.p1"
  - "R.p2 > 0"
```

Listing 6.1: Property file for illegal access to etc/shadow file.

Let's say the property file of Figure 6.1 is stored in a `shadow.yaml` file, and the strace log is stored in a `trace.log` file. Then, the validation of such a property can be executed with the following command:

[5]https://github.com/josephburnett/jd#command-line-usage

```
plogchecker -p shadow.yaml  -l trace.log -s TEXT
```

The final report will be printed to a standard output in text format and also saved in the `report.json` file.

6.3 Cross-Platform Compilation

This section is devoted to testing *Plogchecker 2.0* in terms of functionality on different operating systems with different architectures. The correct functionality of this tool on different platforms is important for the possibility of its use on a wide range of real systems in the context of runtime verification. One can find all combinations of *OS*, *Arch*, *CPU*, and *RAM* on which *Plogchecker 2.0* was tested in Table 6.3. Cross-platform testing was performed on all available unit and integration tests. On all platforms, the tests passed successfully. Based on this, the *Plogchecker 2.0* can be declared compatible across different operating systems with different architectures.

OS	**Arch**	**CPU**	**RAM**
macOS Big Sur (version 11.6.4)	x86_64	2,7 GHz Quad-Core Intel Core i7	16 GB
Windows 10 64-bit (version 10.0.19043)	x64-based	AMD Ryzen 5 3600 6-Core Processor	16 GB
GNU/Linux	x86-64	1,9 GHz Intel Core i7-8650U	16 GB

Table 6.3: Architectures on which *Plogchecker 2.0* was tested.

Chapter 7

Conclusion

This thesis started with a basic description of the individual parts of *runtime verification*. Then it looked more closely at regular expressions and the creation of deterministic finite automatons from them. Since *Plogchecker 2.0* mainly builds on its predecessor, *Plogchecker* [12], one whole chapter is devoted to this tool. This chapter introduces the *Plogchecker* tool and its approximate functioning. At the end of the chapter , its evaluation in terms of computation time and memory consumption is described. The evaluation showed that although *Plogchecker* works well, it is not usable for the verification of real systems for various reasons. Then, the thesis went on to suggest possible enhancements in the framework of *Plogchecker 2.0* that were mainly focused on the monitoring algorithm process. Further, the thesis focused on adding support for event parameter data types. And last but not least, adding support for the discard operator within the bad properties section. Next, an automated garbage collection process was finally proposed, without which, this tool would not be usable as the amount of consumed memory would continuously grow with further incoming events.

The last two sections of this thesis are devoted to the implementation and evaluation details of *Plogchecker 2.0*. In the evaluation section, both unit and integration tests that were used during the implementation of this tool were described. Next, the evaluation section was devoted to comparing *Plogchecker* and *Plogchecker 2.0* against each other in terms of *computation time* and *memory consumption*. The result is that *Plogchecker 2.0* performs better in both metrics. Furthermore, the comparison benchmark showed that thanks to the automatic garbage collecting process, *Plogchecker 2.0* was able to slow down the constant increase of memory usage with each incoming event.

Although *Plogchecker 2.0* has been tested, the tool contains several known issues. The first issue relates to the non-specification of constraints between all possible event pairs of a given property. Suppose we have a property `p1: A B* C`, where each event contains one parameter. Next, suppose some constraint is specified between events `B` and `C`. Further, no constraint is specified over the other possible event pairs. If the input log contained the event sequence `'a.c'`, then the property `p1` would not be satisfied from the point of view of the *Plogchecker 2.0* monitoring algorithm. This is based on the way the `get_starting _tables` function works (see Section 4.1). Another known issue is the lack of support for all possible, lexically correct expressions over event parameters in the constraints section. Currently, only those from the `AvailableLexConfigurations` (Section 5.2 and Appendix A) structure are supported.

Next, one of the many interesting improvements would be to add support for the *JSON data type* as an event parameter. One could then perform many interesting operations over such data type. Possible improvements also include extending the context of a single event to more than one line. Currently, *Plogchecker 2.0* only supports the one row one event strategy. However, this may often not be sufficient since one event can span over multiple lines in many system logs.

All described issues and possible improvements should be the subject of development either in the framework of other diploma theses or directly by the Testos[1] group at the Brno University of Technology.

[1] http://testos.org/

Bibliography

[1] *Regular Expressions.* [Online; accessed 18-February-2022]. Available at: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html.

[2] Barringer, H., Rydeheard, D. and Havelund, K. Rule Systems for Run-Time Monitoring: From Eagle to RuleR. *J Logic Comput.* june 2010, vol. 20. DOI: 10.1093/logcom/exn076.

[3] 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.

[4] Czerwonka, J. Pairwise testing in real world. In: Citeseer. *24th Pacific Northwest Software Quality Conference.* 2006, vol. 200.

[5] Havelund, K. and Peled, D. Runtime Verification: From Propositional to First-Order Temporal Logic. In: Colombo, C. and Leucker, M., ed. *Runtime Verification.* Cham: Springer International Publishing, 2018, p. 90–112. ISBN 978-3-030-03769-7.

[6] Havelund, K., Peled, D. and Ulus, D. First-order temporal logic monitoring with BDDs. *Formal Methods in System Design.* Dec 2020, vol. 56, no. 1, p. 1–21. DOI: 10.1007/s10703-018-00327-4. ISSN 1572-8102. Available at: https://doi.org/10.1007/s10703-018-00327-4.

[7] Jin, D., Meredith, P. O., Griffith, D. and Rosu, G. Garbage Collection for Monitoring Parametric Properties. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery. jun 2011, vol. 46, no. 6, p. 415–424. DOI: 10.1145/1993316.1993547. ISSN 0362-1340. Available at: https://doi.org/10.1145/1993316.1993547.

[8] Lamport, L. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering.* 1977, SE-3, no. 2, p. 125–143. DOI: 10.1109/TSE.1977.229904.

[9] Leucker, M. Teaching Runtime Verification. In: Khurshid, S. and Sen, K., ed. *Runtime Verification.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, p. 34–48. ISBN 978-3-642-29860-8.

[10] Luo, Q., Zhang, Y., Lee, C., Jin, D., Meredith, P. O. et al. RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties. In: Bonakdarpour, B. and Smolka, S. A., ed. *Runtime Verification.* Cham: Springer International Publishing, 2014, p. 285–300. ISBN 978-3-319-11164-3.

[11] Martin, M., Livshits, B. and Lam, M. S. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery. oct 2005, vol. 40, no. 10, p. 365–383. DOI: 10.1145/1103845.1094840. ISSN 0362-1340. Available at: https://doi.org/10.1145/1103845.1094840.

[12] 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/.

[13] Sen, K. and Rosu, G. Generating Optimal Monitors for Extended Regular Expressions. *Proceedings of the 3rd International Workshop on Runtime Verification.* 2003, p. 162–181. DOI: 10.1007/s10009-011-0198-6.

[14] Wikipedia. *Go (programming language) — Wikipedia, The Free Encyclopedia.* 2022. [Online; accessed 12-April-2022]. Available at: http://en.wikipedia.org/w/index.php?title=Go%20(programming%20language)&oldid=1079071629.

[15] Wikipedia. *Python (programming language) — Wikipedia, The Free Encyclopedia.* 2022. [Online; accessed 12-April-2022]. Available at: http://en.wikipedia.org/w/index.php?title=Python%20(programming%20language)&oldid=1082277512.

[16] Češka, M. and Vojnar, T. Theoretical Computer Science. february 2009. Available at: https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FTIN-IT%2Ftexts%2FoporaTCS.pdf&cid=14173.

Appendix A

Grammar Within the Constraints Section

```
"WORD=WORD",
"NUMBER=NUMBER",
"DATESTAMP_RFC1123=DATESTAMP_RFC1123",
"DATE_ISO8601=DATE_ISO8601",
"BOOL=BOOL",

"WORD!=WORD",
"NUMBER!=NUMBER",
"DATESTAMP_RFC1123!=DATESTAMP_RFC1123",
"DATE_ISO8601!=DATE_ISO8601",
"BOOL!=BOOL",

"NUMBER>NUMBER",
"NUMBER<NUMBER",
"NUMBER>=NUMBER",
"NUMBER<=NUMBER",

"DATESTAMP_RFC1123>DATESTAMP_RFC1123",
"DATESTAMP_RFC1123<DATESTAMP_RFC1123",
"DATESTAMP_RFC1123>=DATESTAMP_RFC1123",
"DATESTAMP_RFC1123<=DATESTAMP_RFC1123",
"DATE_ISO8601>DATE_ISO8601",
"DATE_ISO8601<DATE_ISO8601",
"DATE_ISO8601>=DATE_ISO8601",
"DATE_ISO8601<=DATE_ISO8601",

"NUMBER+NUMBER=NUMBER",
"NUMBER+NUMBER!=NUMBER",
"NUMBER+NUMBER>NUMBER",
"NUMBER+NUMBER<NUMBER",
"NUMBER+NUMBER>=NUMBER",
"NUMBER+NUMBER<=NUMBER",

"DATESTAMP_RFC1123+DURATION!=DATESTAMP_RFC1123",
"DATESTAMP_RFC1123+DURATION=DATESTAMP_RFC1123",
"DATESTAMP_RFC1123+DURATION>DATESTAMP_RFC1123",
"DATESTAMP_RFC1123+DURATION<DATESTAMP_RFC1123",
"DATESTAMP_RFC1123+DURATION>=DATESTAMP_RFC1123",
"DATESTAMP_RFC1123+DURATION<=DATESTAMP_RFC1123",
"DATE_ISO8601+DURATION!=DATE_ISO8601",
"DATE_ISO8601+DURATION=DATE_ISO8601",
"DATE_ISO8601+DURATION>DATE_ISO8601",
"DATE_ISO8601+DURATION<DATE_ISO8601",
"DATE_ISO8601+DURATION>=DATE_ISO8601",
"DATE_ISO8601+DURATION<=DATE_ISO8601",

"NUMBER-NUMBER=NUMBER",
"NUMBER-NUMBER!=NUMBER",
"NUMBER-NUMBER>NUMBER",
"NUMBER-NUMBER<NUMBER",
"NUMBER-NUMBER>=NUMBER",
"NUMBER-NUMBER<=NUMBER",

"DATESTAMP_RFC1123-DATESTAMP_RFC1123=DURATION",
"DATESTAMP_RFC1123-DATESTAMP_RFC1123!=DURATION",
"DATESTAMP_RFC1123-DATESTAMP_RFC1123>DURATION",
"DATESTAMP_RFC1123-DATESTAMP_RFC1123<DURATION",
"DATESTAMP_RFC1123-DATESTAMP_RFC1123>=DURATION",
"DATESTAMP_RFC1123-DATESTAMP_RFC1123<=DURATION",
"DATE_ISO8601-DATE_ISO8601=DURATION",
"DATE_ISO8601-DATE_ISO8601!=DURATION",
"DATE_ISO8601-DATE_ISO8601>DURATION",
"DATE_ISO8601-DATE_ISO8601<DURATION",
"DATE_ISO8601-DATE_ISO8601>=DURATION",
"DATE_ISO8601-DATE_ISO8601<=DURATION",
```

Listing A.1: Full contents of *AvailableLexConfigurations* structure.

Appendix B

Memory Consumption of Individual Benchmark Test Cases

Each figure represents a specific test case from Section 3.4, where:

- **P**: Memory consumption of *Plogchecker* over time,
- **P__2**: Memory consumption of *Plogchecker 2.0* over time,
- **P2__garbage**: The time at which the garbage collector was activated automatically within *Plogchecker 2.0*.

The data in these figures are sampled so that the visualization in time is approximately the same for both tools. This is done for better visualization since *Plogchecker 2.0* has shorter run times, and the curves corresponding to it would be almost invisible.

Test 01

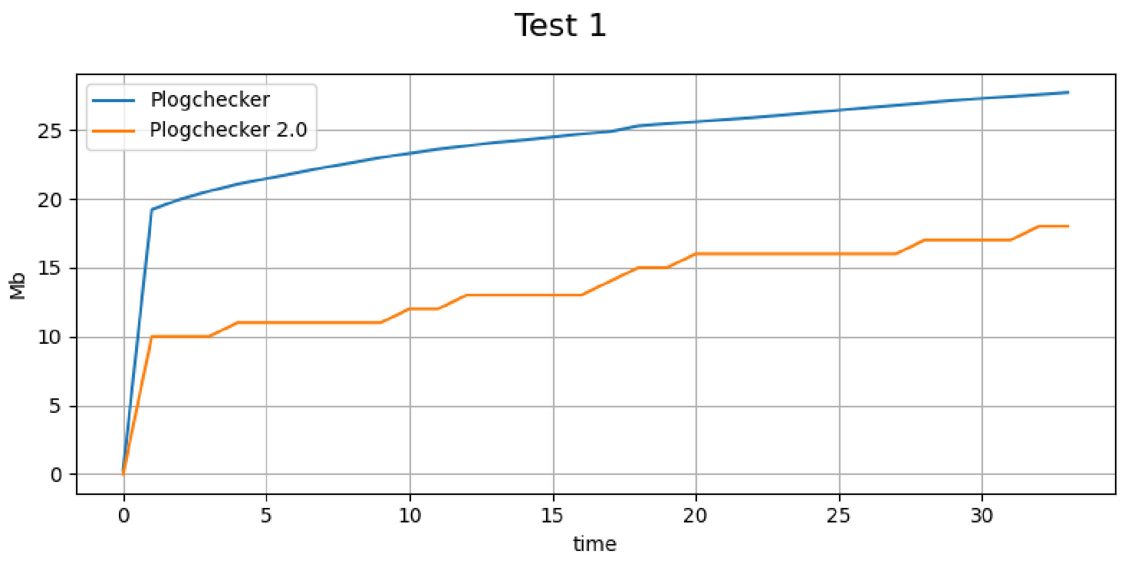


Figure B.1: Memory consumption of Plogchecker and Plogchecker 2.0 within test case 01 over period of time.

Test 02

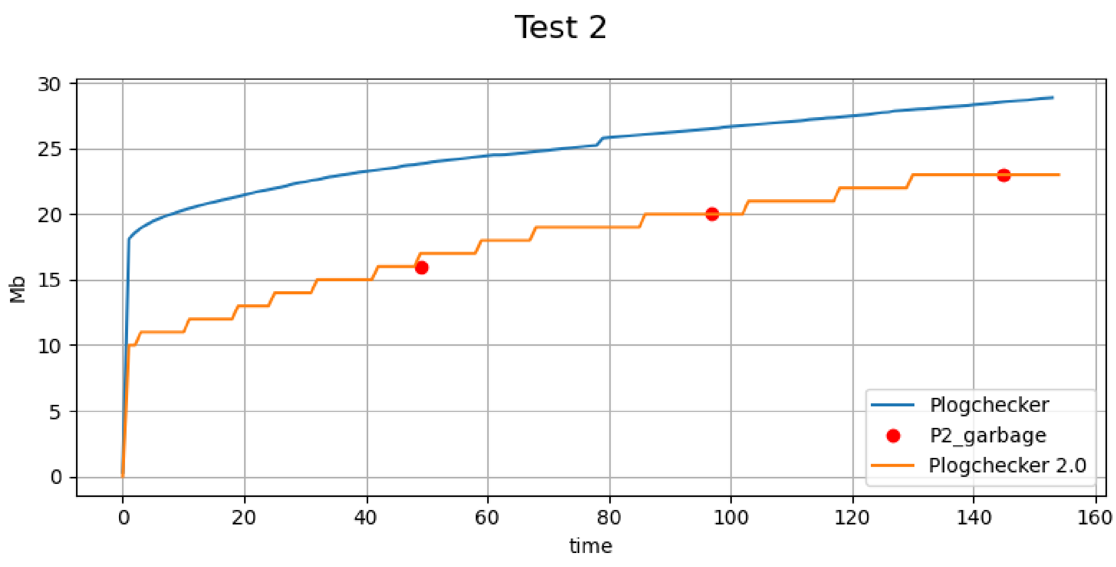


Figure B.2: Memory consumption of Plogchecker and Plogchecker 2.0 within test case 02 over period of time.

Test 03

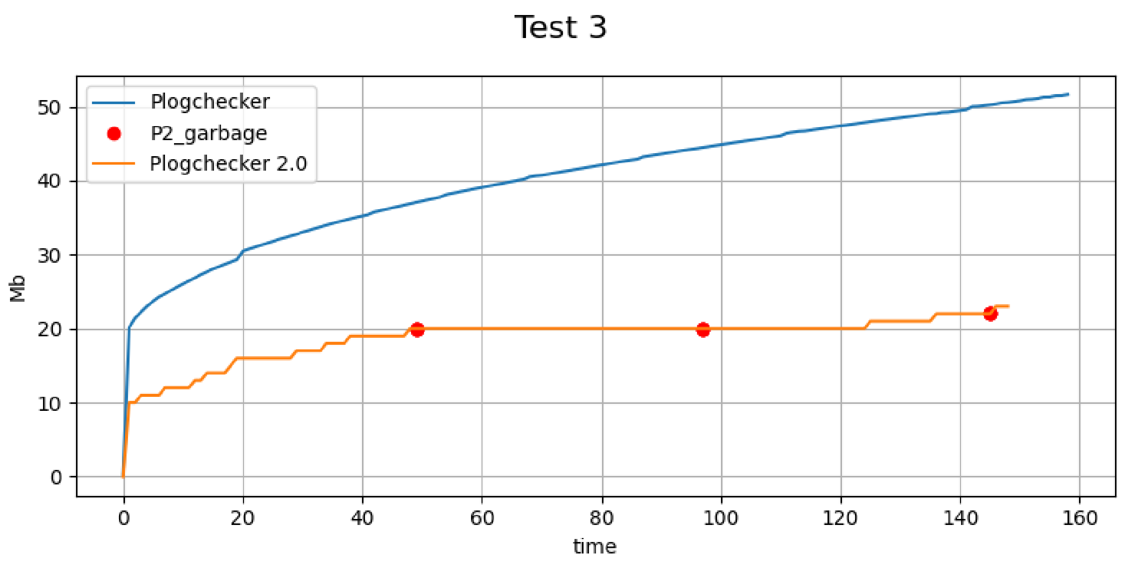


Figure B.3: Memory consumption of Plogchecker and Plogchecker 2.0 within test case 03 over period of time.

Test 04

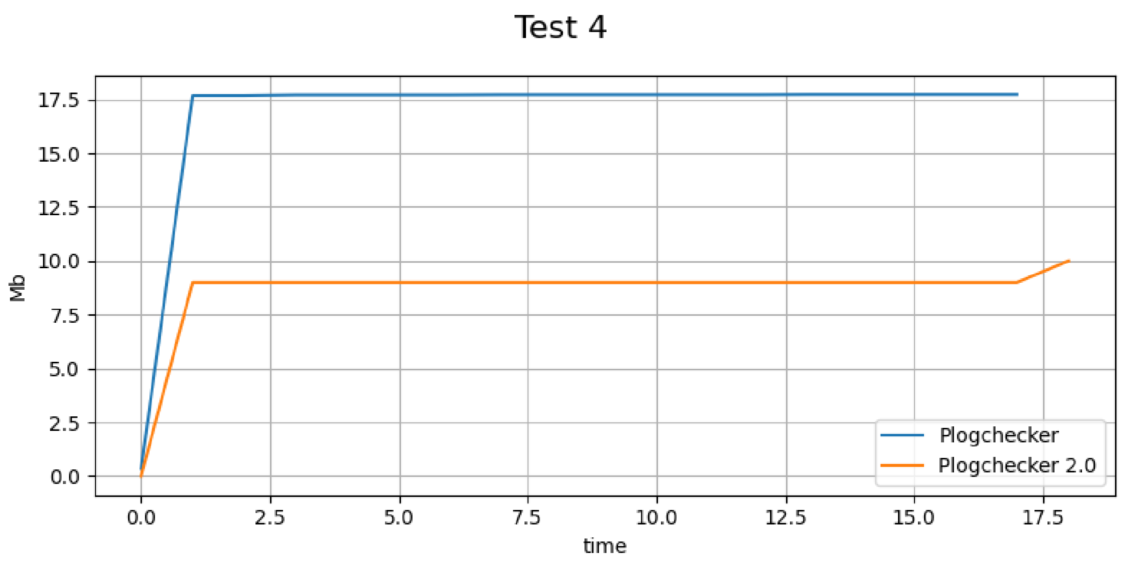


Figure B.4: Memory consumption of Plogchecker and Plogchecker 2.0 within test case 04 over period of time.

Test 05

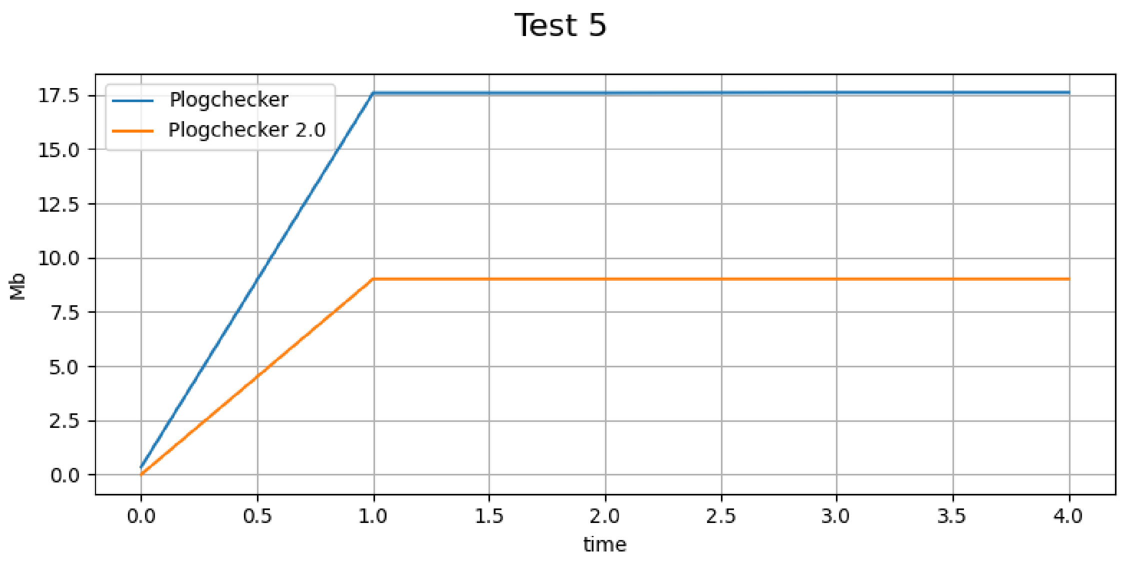


Figure B.5: Memory consumption of Plogchecker and Plogchecker 2.0 within test case 05 over period of time.

Test 06

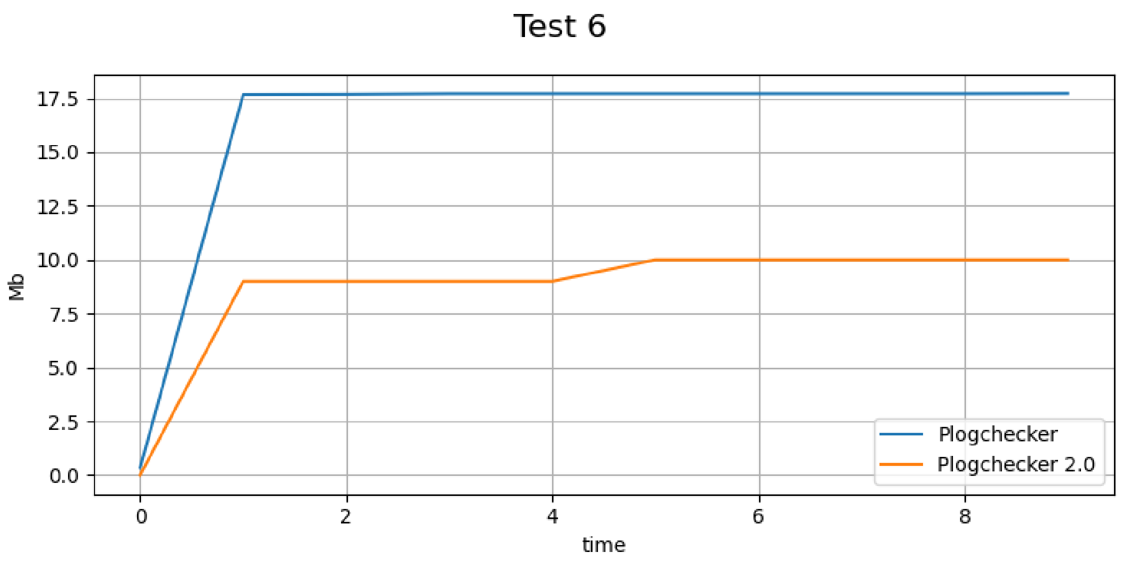


Figure B.6: Memory consumption of Plogchecker and Plogchecker 2.0 within test case 06 over period of time.

Test 07

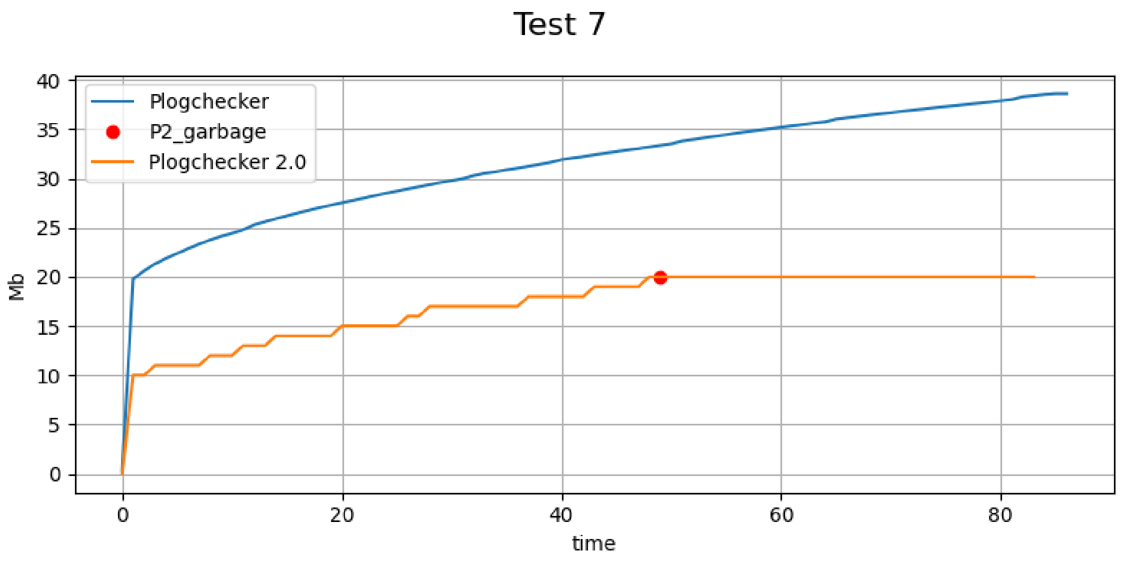


Figure B.7: Memory consumption of Plogchecker and Plogchecker 2.0 within test case 07 over period of time.

Test 08

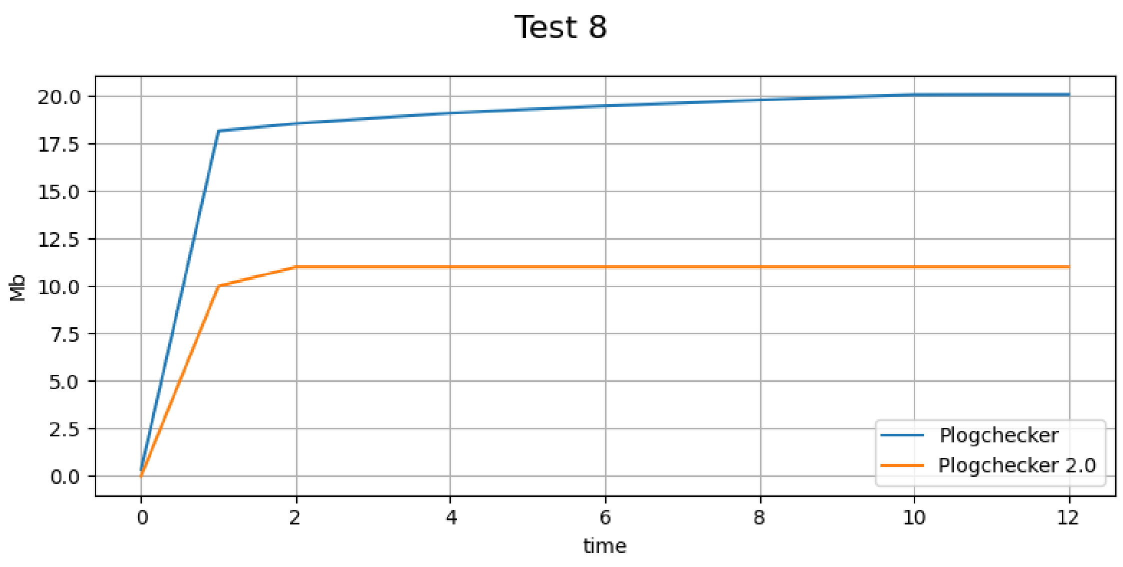


Figure B.8: Memory consumption of Plogchecker and Plogchecker 2.0 within test case 08 over period of time.

Test 09

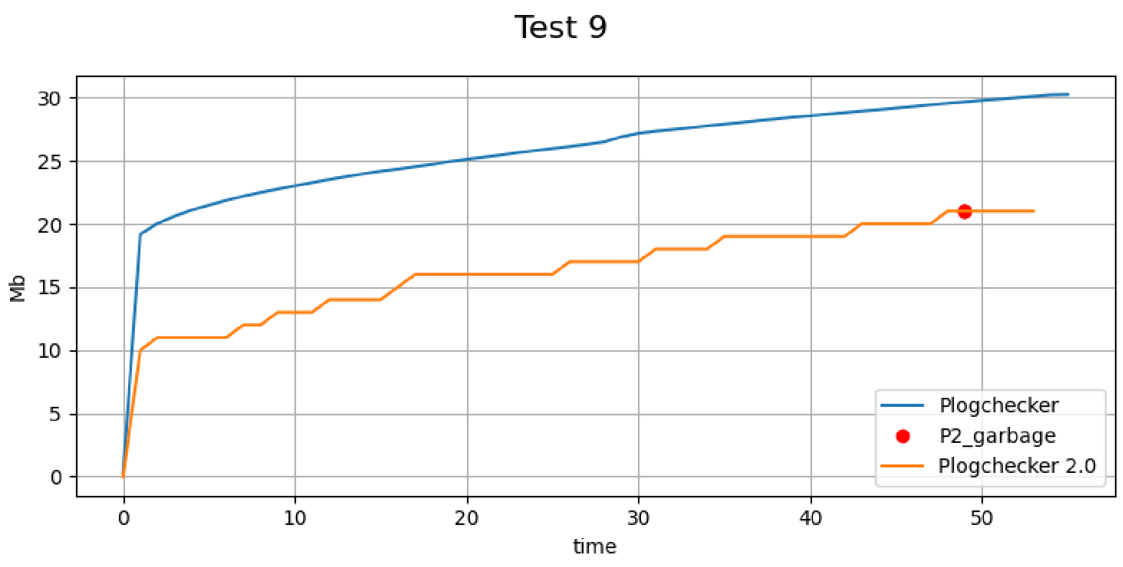


Figure B.9: Memory consumption of Plogchecker and Plogchecker 2.0 within test case 09 over period of time.

Test 10

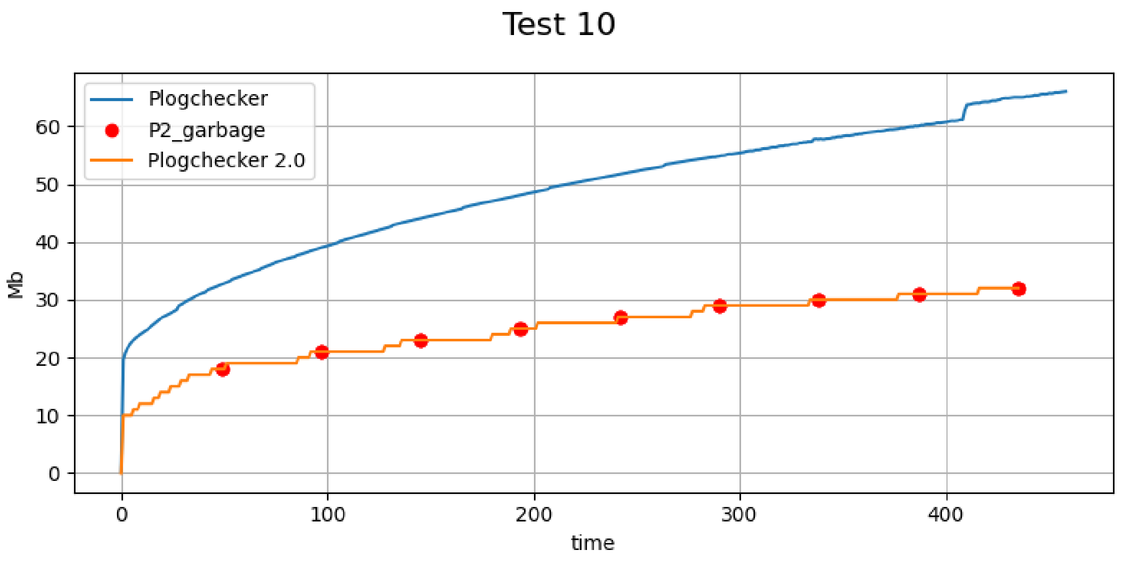


Figure B.10: Memory consumption of Plogchecker and Plogchecker 2.0 within test case 10 over period of time.

Test 11

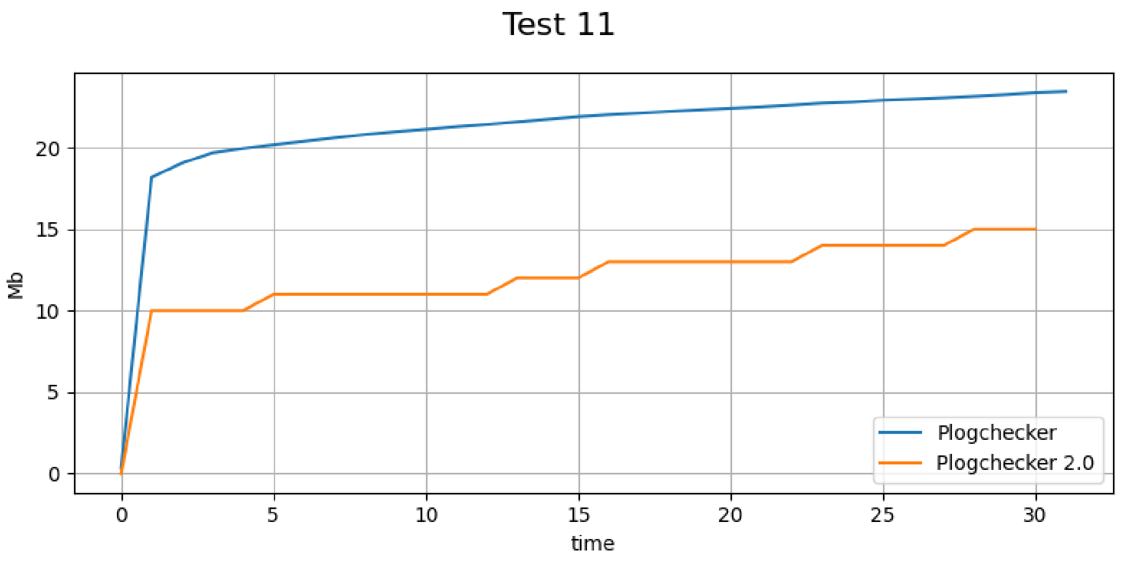


Figure B.11: Memory consumption of Plogchecker and Plogchecker 2.0 within test case 11 over period of time.

Appendix C

Contents of the Attached Memory Media

This appendix lists the contents of the attached memory media. In particular, the attached memory media contains the following:

- **xcalad01-thesis-2022.pdf** - This thesis in PDF format.
- **thesis-latex** - The source code of this thesis.
- **plogcheckerng** - The source code of the *Plogchecker 2.0*. Apart from the source code of this tool, the folder contains data and scripts for running the benchmarks, described in Section 6.1, in the `benchmarks/` folder. A practical example for *strace* log and property is also available in the `examples/` folder.

Appendix D

Installation and User Manual

This appendix serves as an installation and user manual. A more detailed description can be found on the *Plogchecker 2.0 Gitlab repository* (`https://pajda.fit.vutbr.cz/testos/plogcheckerng/-/tree/main`) or in the attached memory media (Appendix C).

Before installing and using *Plogchecker 2.0*, it is necessary to install the Go programming language with version at least `1.18`. One can find the specific installation instructions on its official web page[1]. Moreover, it is assumed that the current working directory contains all files from the `plogcheckerng` directory described in Appendix C. If not, the directory can be cloned using the `git` command through `SSH` or `HTTPS` as follows:

```
# SSH
git clone git@pajda.fit.vutbr.cz:testos/plogcheckerng.git
# HTTPS
git clone https://pajda.fit.vutbr.cz/testos/plogcheckerng.git
```

Installation Manual

Plogchecker 2.0 can be installed in the two following ways:

- *make* command: `make` or `make build`,
- *go* tool: `go build .`

The result of the installation process is a binary `plogchecker`, which can be used without any other dependencies. A description of how to use this binary can be found in the following section.

User Manual

This section assumes that the `plogchecker` binary is already installed. If not, check the section above that describes how to install this binary. *Plogchecker 2.0* expects two main inputs, the properties file and the log of the monitored system. The property file must be specified in YAML format and must meet all conditions from Section 5.1. Furthermore, one can specify the log of the monitored system either by the specific file that contains this log or by standard input. After running the `plogchecker` binary over such inputs, a report about violations of monitored properties will be stored in the `report.json` file.

[1] `https://go.dev/doc/install`

Step By Step Startup Example

This section serves as a step-by-step guide on using a `plogchecker` binary with the specific property file and input log.

Step 1 – Property File

Consider the property file, `properties.yaml`, from Listing D.1 that specifies one good property `p1` and one bad property `p2`. Property `p1` expects two events, `A` and `B`. Each event carries one parameter of type `NUMBER`, where these two parameters must be equal. Property `p2` expects only one event `C`, which carries one `WORD` type parameter.

```yaml
properties:
  p1: "A B"
bad_properties:
  p2: "C"
events:
  A: "a %{NUMBER:p1}"
  B: "b %{NUMBER:p1}"
  C: "c %{WORD:p1}"
constraints:
  - A.p1 = B.p1
```

Listing D.1: Property file example.

Step 2 – Input Log

Consider the following input log, `trace.log`, from Listing D.2.

```
a 1
c word
a 2
b 1
```

Listing D.2: Input log example.

Step 3 – Running Plogchecker

The `plogchecker` binary can be then executed as follows:

- `./plogchecker -l trace.log -p properties.yaml`

Step 4 – Final Report

Listing D.3 shows the violated monitoring properties after running the `plogchecker` binary with `properties.yaml` and `trace.log` inputs. As you can see, the input log violated property `p1` just once. The sequence of events that violates this property is made only by event `A` which is located on line number `3` of the input log. Property `p1` is violated because event `B` with parameter `2` did not occur after this event. Property `p2`, like property `p1`, is violated just once. Since it is a bad property, its violation is considered to be the fulfillment of the given event sequence. Thus, as can be seen, property `p2` was violated by event `C`, which is located on line number `2` of the input log.

```
{
  "properties": {
    "p1": {
      "property": "A B",
      "violated": [
        [
          {
            "eventId": "A",
            "lineNo": 3,
            "lineContent": "a 2"
          }
        ]
      ]
    }
  },
  "badProperties": {
    "p2": {
      "property": "C",
      "violated": [
        [
          {
            "eventId": "C",
            "lineNo": 2,
            "lineContent": "c word"
          }
        ]
      ]
    }
  }
}
```

Listing D.3: Final report in JSON format after executing plogchecker binary with properties.yaml and trace.log inputs.

Input Arguments

The following input arguments can be used when running `plogchecker` binary:

`-p {filename}`

- Property file in *YAML* format. If not specified, the default value is *logproperties.yaml*.

`-l {filename}`

- File that contains the log of the monitoring system. If not specified, the default value is input from *stdin*.

`-r {directory}`

– Specifies the directory in which to store the file that contains the final report of the monitoring algorithm (`report.json` file). If not specified, the default value is *current directory*.

`-s [JSON|TEXT]`

– Specifies the streaming format of bad properties violations. If specified, the valid values are *JSON* or *TEXT*. The bad properties violations are stored to final report as well. The default value is empty string (`""`), which means that streaming of bad properties violations is disabled.

`-m [true|false]`

– Enables (true) or disables (false) printing out the current memory allocated in MB to stdout. If not specified, the default value is false.