

Toward a Taxonomy of Typestate Systems: Soundness Without Alias Information

by

Billy Bai

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2025

© Billy Bai 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

External third-party APIs are commonly used in real-world programming. Specifications or protocols are given by the designer to direct the usage of those APIs combined. The violation of specifications may crash the program, or weaken or break the guarantees of the framework, which would be vital to some safety-critical systems such as encryption.

Typestate system is one of the approaches to check the valid use of APIs based on a given protocol. However, the soundness of the general typestate system relies on the soundness of external alias information. Obtaining whole-program alias information is often expensive and inefficient. One way to remove this dependency is to restrict the protocols. This work tries to examine the class of protocols that can be soundly checked without alias information. We formalize the notion of the protocol and the typestate system, from where we further construct some special classes of protocols. We identify the group of typestate systems called the accumulative typestate system, which is a superset of typestate systems that can be checked soundly without alias information. We also pin down two groups of accumulative typestate systems that are actually sound without alias information.

Acknowledgements

First of all, I would like to thank all the people who made this thesis possible.

I would like to express my deepest gratitude to my supervisor, Professor Arie Gurfinkel, for accepting the initially unknowing me as his student and guiding me with patience throughout my journey. His support and mentorship have been invaluable to both my academic and personal growth.

I would like to thank Professor Werner Dietl for his help throughout my Master's life. The feedback from him always accurately reveals the flaws.

Finally, I would also like to thank my lab mates and research companions — Aosen Xiong, Joseph Tafese, Siddharth Priya, and Yusen Su — for their generous help and warm companionship, both in research and in everyday life.

Table of Contents

Author's Declaration	ii
Abstract	iii
Acknowledgements	iv
List of Figures	vii
1 Introduction	1
2 Background	3
2.1 Formal Language	3
2.2 Regular Language	5
2.2.1 Formal Definition	5
2.2.2 Regular Expression	5
2.2.3 Derivative of Regular Expression	6
3 Typestate System	8
3.1 Language	8
3.1.1 Protocol	8
3.1.2 Syntax	12
3.1.3 Operational Semantics	14
3.2 Typestate System	18
3.2.1 Alias Information	18
3.2.2 Typing Rules	19

4	Pumping Closure	24
4.1	Pumping Closure	24
4.2	Special Pumping Closures	32
4.2.1	Prefix-include Pumping Closure	33
4.2.2	Permutation-extended Pumping Closure	37
5	Accumulative Typestate System	42
5.1	Motivating Example	42
5.2	Accumulative Typestate System	45
5.2.1	Definition	45
5.2.2	Need of Alias Information	46
5.3	Special Accumulative Typestate System	48
5.4	Discussion	50
6	Related Work	51
6.1	Other Attempts to remove alias information	51
6.1.1	Heap-Monotonic Typestate	51
6.1.2	Accumulation Analysis	52
6.1.3	Linear and Uniqueness Typing	53
6.1.4	Capabilities-Based Typestate Systems	54
6.1.5	Separation Logic-Based Approaches	55
6.2	Typestate Systems in Other Form	56
6.2.1	Finite State Machines (FSMs) and Deterministic Finite Automata (DFAs)	56
6.2.2	Transition Relations	57
6.3	Other Behavioral Types	57
6.3.1	Session Types	57
6.3.2	Effect Types	58
7	Conclusion	60
	References	61

List of Figures

3.1	A builder class with methods set and build, where the protocol is written in comment.	9
3.2	DFA of the protocol for Build class.	10
3.3	A simple program using the builder.	11
3.4	Syntax of CHECKOL	13
3.5	A file class with open, write and close method.	14
3.6	Runtime model of CHECKOL	15
3.7	Operational semantics of CHECKOL for expression.	16
3.8	Operational semantics of CHECKOL for statement.	17
3.9	Typing rules of the typestate system over the language.	20
4.1	DFA of example 6	31
5.1	Builder for Car.	43
5.2	Compiler Front-end.	44
6.1	DFA of File Object	56

Chapter 1

Introduction

Typestate systems provide an important mechanism for statically verifying program correctness by tracking and enforcing permissible sequences of operations (mainly method calls) on objects according to predefined state protocols, which form a flow-sensitive type qualifier system [20]. Initially introduced by Strom and Yemini [50, 49] as a programming language concept intended to enhance software reliability, typestate has evolved significantly and found applications in a variety of domains, including operating systems [37, 36], software security [30, 1, 7], and blockchain smart contracts [10]. The essential innovation of typestate analysis lies in its ability to detect erroneous state transitions at compile time, thereby preventing critical runtime failures.

Typestate-oriented programming languages, like Plural [7], Plaid [1], Papaya [29], and Obsidian [10], embed typestate directly into their language semantics, significantly reducing boilerplate code and improving developer productivity and correctness. Obsidian, in particular, illustrated how typestate and asset-tracking enhance safety in blockchain programming by preventing misuse of sensitive resources, reducing the risk of critical vulnerabilities.

One key application of typestate systems is their ability to mitigate or prevent integration bugs [44]. By statically verifying compliance with usage protocols, typestate systems ensure that interactions between independently developed modules adhere to specified contracts, thereby mitigating common integration issues like improper API usage and unintended resource sharing. The practical benefits of typestate verification in reducing such errors have been demonstrated extensively in both industrial and academic contexts [24, 15, 4, 23].

Despite these advantages, practical adoption of typestate analyses has historically been hindered by the complexity arising from aliasing and scalability concerns. Aliasing [48] complicates reasoning about object states because multiple references to the same object can lead to ambiguous state transitions. Earlier attempts to handle aliasing often imposed strict limitations, restricting usability in realistic programs [13].

To address the challenge of aliasing, subsequent research introduced flexible mechanisms such as view-based alias control [39, 46], access permissions [6], and

alias-summary-based approaches [19, 52, 36, 37] that restrict the occurrence of aliases.

Some other researcher took another path to identify the subset of protocols being sound without alias information. Some recent work [33] about accumulation analysis proposes a formalized typestate system that eliminates the dependency on alias information.

In this thesis, we take inspiration from the restrictions of protocols and propose a formalization that takes on a lightweight, yet expressive typestate system described by regular expressions [11] to define state protocols. Then an accumulative typestate system is built up from this framework by restricting the protocols, which is shown to be a superset of typestate systems that can be checked soundly without alias information. Intuitively, the restriction originates from the idea that if an operation is done, it is always valid to be applied again. In other words, the access to operations can only be gained, but not lost. We further pin down two classes of accumulative typestate systems that are definitely sound without alias information, which provides a cheap yet accurate system to check the compliance of protocols.

Our contributions are:

- a formalization of protocol and its conformance criteria, a language CHECKOL with the ability to check the conformance of protocol, and a general typestate system with alias information (Chapter 3);
- a formal definition of pumping closure language that defines the accumulative property while preserving the regularity, interleaving shuffle that encodes the interactions between aliases, two subgroup of pumping closure that is closed under interleaving shuffle with proofs (Chapter 4);
- a formal definition of accumulative typestate system with the proof that it is a superset of the typestate system being sound without alias information, and two subgroup of accumulative typestate system with the proof that they are subset of the typestate system being sound without alias information (Chapter 5);
- a literature survey about the related work (Chapter 6).

The thesis is organized as follows: The background introduction about regular expression and its derivative is given in Chapter 2. The CHECKOL with general typestate system is presented in Chapter 3. Chapter 4 gives the language level restriction on protocols with proofs of soundness. The language level restriction is then added to the typestate system in Chapter 5. Finally, Chapter 6 connects this work to related works.

Chapter 2

Background

In this chapter, we introduce background knowledge of formal language, regular language, regular expression, and derivative. They are later used to describe the protocol we want to check in the program.

2.1 Formal Language

In this part, we introduce the fundamental structure of our work. A formal language L is a set of *words* s over some *alphabet* Σ . An alphabet can be an arbitrary finite set whose elements a are called *letters*. A word is a sequence of letters from the alphabet. In particular, the empty word is denoted as ε .

For convenience, we also have the following notation.

- The length of the word s is denoted by $|s|$, which is equal to the number of letters in s .
- The i th letter of a word is denoted as $s[i]$.
- The subsequence of a word s from the i th letter to the j th letter is denoted as $s[i..j]$. Especially, we set $s[i..j] = \varepsilon$ when $i > j$.
- The occurrence of a letter a in a word s as $a \in s$.

A simple example of a formal language is as follows: Given an alphabet $\Sigma = \{a, b\}$, a language over Σ could be $L = \{a, aaa, ab, abab, \varepsilon\}$.

There are different operations over words and languages. First, we introduce the only operation over words, which is concatenation.

Definition 1 (Concatenation). Given a formal language L over an alphabet Σ . The concatenation of two words $s_1 = a_1a_2 \cdots a_m$ and $s_2 = b_1b_2 \cdots b_n$ is defined as $s_1 \cdot s_2 = a_1 \cdots a_mb_1 \cdots b_n$.

The concatenation is important in constructing words, since every word can be seen as the result of a sequence of concatenations of single letters. That is, a word can also be seen as a sequence of words. Therefore, we omit the symbol of $\cdot$. For example, $s_1 \cdot s_2$ is written as $s_1 s_2$.

There are also operations over languages. Here, we define them at the set level.

Definition 2 (Formal Language Operation). Given two formal languages L and L' over alphabets Σ_1 and Σ_2 . We have

1. *Addition*: $L + L' = L \cup L'$ over $\Sigma_1 \cup \Sigma_2$.
2. *Concatenation* (or *multiplication*): $L_1 \cdot L_2 = \{sl \mid s \in L \wedge l \in L'\}$ over $\Sigma_1 \cup \Sigma_2$.
3. *Kleene star*: $L^* = \sum_{n=0}^{+\infty} L^n$ over Σ_1 , where $L^0 = \{\varepsilon\}$ and $L^{n+1} = L^n L$.

The alphabet changes are often omitted for convenience if they are not mentioned. The addition of regular expressions and union of languages are used interchangeably in the rest of this thesis. Here we also present some properties that will be used in the later proofs, and the proofs of them are also omitted.

Proposition 1. The addition operation has the following properties:

1. *Exchangeability*: Given two formal languages L_1 and L_2 , we have $L_1 + L_2 = L_2 + L_1$.
2. *Associativity*: Given three formal languages L_1, L_2 and L_3 , we have $L_1 + L_2 + L_3 = L_1 + (L_2 + L_3)$.
3. *Idempotency*: Given a formal language L and a word $s \in L$, we have $s + L = L$.

Additionally, note that the Kleene star of the alphabet is exactly the language of all possible words over this alphabet. Therefore, we often use Σ^* to denote this largest language over the alphabet Σ .

Finally, we introduce the concept of prefix.

Definition 3 (Prefix). Given a word $s \in \Sigma^*$, a word $l \in \Sigma^*$ is called a prefix s iff

$$\exists w \in \Sigma^* . s = lw$$

By definition, the prefixes of a given word s can be categorized into three different types. They are 1) ε for empty word where $w = s$, 2) the whole origin word s where $w = \varepsilon$ and 3) any sub-sequence starting from the first letter $s[1..k]$ where $k = 1, \dots, |s| - 1$ and $w = s[k + 1, |s|]$. We conclude this into a proposition.

Proposition 2. Given a word $s \in \Sigma^*$, we have

$$\text{pref}(s) = \{\varepsilon\} \cup \{s[1..k] \cdot k = 1, \dots, |s|\}$$

2.2 Regular Language

Here we introduce a special class of formal languages called regular languages.

2.2.1 Formal Definition

Definition 4 (Regular Language). A language $L \subseteq \Sigma^*$ is *regular* if:

1. $L = \emptyset$, or
2. $L = \{\varepsilon\}$, or
3. $L = \{a\}$, where $a \in \Sigma$, or
4. $L = R_1 R_2$ or $L = R_1 + R_2$ or $L = R_1^*$, where $R_1, R_2 \subseteq \Sigma^*$ are regular language.

From the above definition, we observe that any regular language can be constructed by a finite number of operations over the basic regular languages. The basic regular languages are the empty set, the empty word, and the set of single letters. Therefore, we introduce a syntactic representation of regular languages, which is a regular expression.

2.2.2 Regular Expression

Definition 5 (Regular Expression). Given an alphabet Σ , a regular expression is a syntactic notation used to describe a regular language. The syntax and the semantics (i.e. corresponding language, denoted as $L(s)$) can be defined recursively as follows: • base case:

1. $\emptyset$ is a regular expression representing empty language, i.e. $L(\emptyset) = \emptyset$.
2. ε is a regular expression that represents an empty word, i.e. $L(\varepsilon) = \varepsilon$.
3. $a \in \Sigma$ is a regular expression that represents a single letter of the alphabet, that is, $L(a) = a$.

• recursive case:

1. $r_1 + r_2$ is a regular expression representing the union of languages, i.e. $L(r_1 + r_2) = L(r_1) + L(r_2)$.
2. $r_1 \cdot r_2$ is a regular expression that represents the concatenation of languages, i.e. $L(r_1 \cdot r_2) = \{s_1 s_2 \mid s_1 \in L(r_1) \wedge s_2 \in L(r_2)\}$.
3. r^* is a regular expression that represents the Kleene star of language, i.e. $L(r^*) = (L(r))^* = \sum_{i=0}^{+\infty} L^i(r) = \lim_{k \rightarrow \infty} \sum_{i=0}^k L^i(r)$.

A regular expression over an alphabet Σ is a sequence of characters that defines a search pattern. Basic operations in regular expressions include concatenation, union (denoted by $+$), and the Kleene star (denoted by $*$), which allows the repetition of the preceding element zero or more times.

In this thesis, for convenience, we interchangeably use the regular expression as both a symbolic expression and the corresponding language. For example, we would write $w \in s$ for some word w and regular expression s to represent $w \in L(s)$.

A useful result of regular expression is that the existence of regular expression of a language indicates its regularity. In other words, the regularity of a given language can be reduced to two steps. The first step is to form a regular expression, while the second step is to show that the regular expression represents the same language as the target language.

2.2.3 Derivative of Regular Expression

Here, we use denotational semantics to describe the derivative of regular expressions [9]. Informally, the derivative of a regular expression R w.r.t. a letter l is another regular expression that contains all the words that are accepted by R after preceding with the letter l . There are two functions needed to define this derivative, which are the final function and the derivative function. We start with the first one, called the final function.

Definition 6. The *Final Function* E is a function that maps a regular expression R to the symbols $\{\emptyset, \varepsilon\}$, where $E(r) = \varepsilon$ if the ε is in r and $E(r) = \emptyset$ otherwise.

The final function is used to determine whether the empty word is in the regular expression. This is useful in derivative to deal with the infinite language. With the above definition, we give a recursive way to compute the final function.

Proposition 3. The final function E can be computed as follows:

$$\begin{aligned} E(\emptyset) &= \emptyset \\ E(\varepsilon) &= \varepsilon \\ E(a) &= \emptyset \\ E(R_1 R_2) &= E(R_1) + E(R_2) \\ E(R_1 + R_2) &= E(R_1) E(R_2) \\ E(R^*) &= \varepsilon \end{aligned}$$

Here we use concatenation and addition in the computation based on a simple property of regular expression, which is $\varepsilon\emptyset = \emptyset\varepsilon = \emptyset$ and $\varepsilon + \emptyset = \emptyset + \varepsilon = \varepsilon$.

Now we also give a general definition of derivative and then give an alternative, recursive definition of derivative function utilizing the final function.

Definition 7. The *Derivative Function* D is a function that maps a regular expression R and a letter l to another regular expression such that $D_l(R) = \{w \cdot lw \in R\}$.

Now, we also show a recursive way to compute the derivative function.

Proposition 4. The derivative function D w.r.t. letter l can be computed as follows:

$$\begin{aligned}
 D_l(\emptyset) &= \emptyset \\
 D_l(\varepsilon) &= \emptyset \\
 D_l(a) &= \begin{cases} \emptyset & \text{if } a \neq l \\ \varepsilon & \text{if } a = l \end{cases} \\
 D_l(R_1R_2) &= (D_l(R_1)R_2) + (R_1D_l(R_2)) \\
 D_l(R_1 + R_2) &= D_l(R_1) + D_l(R_2) \\
 D_l(R^*) &= D_l(R)R^*
 \end{aligned}$$

The proof of these two propositions is omitted. They can be proven by induction.

Chapter 3

Typestate System

In this chapter, we introduce the general idea of a typestate system. Like most type systems, typestate systems and analysis target some object-level properties and lift them to the reference level to offer the sound guarantee for the target from the reference-level type checking. The object-level property that the typestate system focuses on is the method invocation. The specification describing the valid sequence of method invocations is encoded by a regular language. Intuitively, a protocol is *conformed* by a sequence of method invocations in a full program execution if this sequence is accepted by the protocol. The typestate system is the kind of type system aiming to statically check the protocol conformance. Since the method invocations history is built up throughout the execution of instructions, the general typestate systems require alias information to improve the precision.

In Section 3.1, we give a formal definition to the kind of protocol checked and the language used to build the typestate system upon. Then, we introduce the general form of the typestate system with alias information in Section 3.2.

3.1 Language

In this section, we introduce the way to describe the conformance of the protocol and present a toy Object-Oriented (OO) language extended with typestate. The language has no loop or conditional statement for simplicity without losing the essence of typestate. In the following, we introduce the syntax of this language and the operational semantics with a runtime model that takes the typestate of the object into account to check if the protocol is enforced.

3.1.1 Protocol

We give a more strict description of the kind of *protocol* we are talking about. In this thesis and later sections, we refer to protocol as the kind of protocol that can be expressed as a finite-state transition system, where the transition is made by method

calls. The restrictions could be relaxed, but it works fine in the target problem we are working on, which is to regulate the calls of the methods. The protocols regulate the use of methods abstractly. They only ensure the methods are called in the right order. In this way, the methods are more like events happening without any affect to the state other than the method call records. The consistency between the actual method implementations and the specification that protocol encodes is beyond the scope of this work. We leave it for future work.

Here is an example of such a protocol.

Example 1. Consider a simple builder *Builder* for some class *C* as shown in Fig. 3.1. The protocol we want to enforce is to make sure that *build()* is called only after the *set()* has been called.

Ignoring some edge cases for simplicity, this protocol can be described as the regular expression ss^*b where *b* stands for *build()* and *s* stands for *set()*. Or alternatively, we can describe it as a deterministic finite automaton (DFA) shown in Fig. 3.2.

```

Class C{ ... }

Class Builder{
    Builder() {...}
    void set(){...}

    // build() should be only called after set has been called at least
    // once.
    C build(){
        return new C(...);
    }
}

```

Figure 3.1: A builder class with methods *set* and *build*, where the protocol is written in comment.

Since the protocol can be expressed as a DFA, it is essentially a regular language where each letter represents one call to a specific method. Thanks to the in-depth study of the regular language, we also have multiple ways to formally describe the protocol. As we have mentioned before, we describe it as an DFA [14, 30, 19], or only by specifying the transition relations [2, 21]. Inspired by previous works [11, 43], regular expression is what we choose to express the protocols in this thesis.

Definition 8. A *regular protocol* of a class *C* is a tuple $\pi_C = (\mathbb{M}, L)$ where $\mathbb{M} : \overline{M} \mapsto \Sigma \cup \{\varepsilon\}$ is a mapping from the defined methods $\overline{M}$ in class *C* to the union of alphabet Σ and ε , *L* is the regular language over the alphabet Σ .

Intuitively, the regular protocol contains a map from the sequence of methods called during an execution to a word, and a language of such words that is allowed by this protocol. We refer to the regular protocol as protocol in the rest of this paper. Also, we use regular expression *r* to represent the regular language *L*.

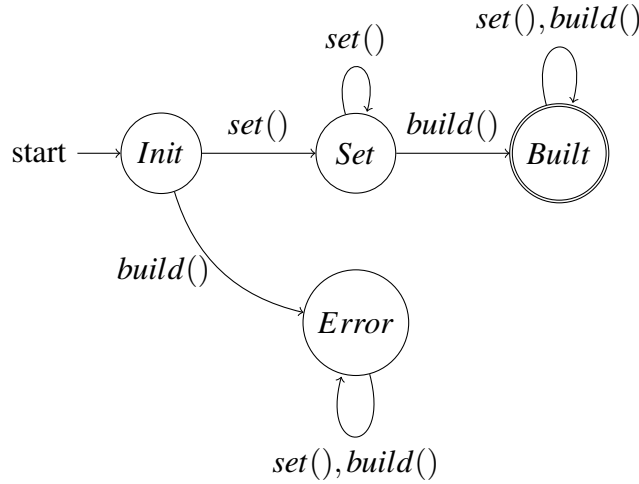


Figure 3.2: DFA of the protocol for Build class.

Definition 9. Let o be an object of class C with defined method $\overline{M} = \{m_1, m_2, \dots, m_n\}$ and a program P . A *history* of o is a finite ordered sequence

$$h_o = (m_{i_1}, m_{i_2}, \dots, m_{i_k})$$

where $i_j \in [1, n]$ for $1 \leq j \leq k$.

Intuitively, each history of o would represent a sequence of methods called on the object o at some point during some execution. In particular, we denote the final history of o as f_o , representing the complete history of o at termination. Note that we limit the sequence to be finite because the non-terminated execution is out of the scope of this thesis.

The history captures all the information about o we need to define the protocol conformance. In this way, the history serves as an abstract but sufficient representation of the execution on object o . Hence, the history can be used to define the conformance of a protocol.

Definition 10. Given a class C with defined method $\overline{M}$, an object o of class C , and a protocol of C $\pi_c = (\llbracket \cdot \rrbracket, r)$, we say a final history of some object o conforms to the protocol π_c , denoted as $f_o \models \pi_c$ if and only if the translated word under $\llbracket \cdot \rrbracket$ is in r , i.e.

$$\llbracket f_o \rrbracket := \llbracket m_{i_1} \rrbracket \cdot \llbracket m_{i_2} \rrbracket \cdots \llbracket m_{i_k} \rrbracket \in r$$

The final history summarizes the behaviors of the object o in some execution. The protocol is conformed when the behaviors of the object o , which is the translated word $\llbracket f_o \rrbracket$, are accepted by the protocol. Here we give a concrete example about history and protocol conformance.

Example 2. Consider the **builder** example. We have mentioned the protocol, which can be described as $\pi_{Builder} = (\llbracket \cdot \rrbracket, ss^*b)$ where $\llbracket \cdot \rrbracket := \{build \mapsto b, set \mapsto s\}$. Fig. 3.3 shows a program that uses the methods from the *Builder* class.

```

Builder GoodBuilder = new Builder();
Builder BadBuilder = new Builder();

GoodBuilder.set();
GoodBuilder.build();
BadBuilder.build();

```

Figure 3.3: A simple program using the builder.

There are two builder objects in a straight line program. Both objects *GoodBuilder* and *BadBuilder* have only one possible final history. *GoodBuilder* has a final history $f_g = (set, build)$ while *BadBuilder* has a final history $f_b = (build)$. We can hence compute the translated word as $\llbracket f_g \rrbracket = sb$ and $\llbracket f_b \rrbracket = b$. Note that we have $\llbracket f_g \rrbracket \in ss^*b$ and $\llbracket f_b \rrbracket \notin ss^*b$. By the Def. 10, $f_g \models \pi_{Builder}$ and $f_b \not\models \pi_{Builder}$.

However, we should notice that this definition is one single check at the end of the execution. In other words, the check requires the final history of o acquired at the end of execution. Next, we show that, actually, we can detect some violations earlier.

The key idea, inspired by [11], is to utilize the *derivative* of a regular expression. The derivative of a language computes a new language of all the suffixes of a given letter in the original language. To get the final history of objects, meta-information is needed to store the histories of all objects and get updated by every method call. The final history is therefore built up incrementally. Each intermediate history is therefore a prefix of the final history. Therefore, starting from the first method call, we can compute the derivative over this called method to compute a new language of all the possible suffixes in the previous language. This idea leads to the following lemmas.

Lemma 1. *Let C be a class with defined methods $\overline{M}$, $\pi_c = (\llbracket \cdot \rrbracket, r)$ be a protocol of C , and o an object of class C . For a final history $f_o = (m_{i_1}, \dots, m_{i_k})$, the derivative of r over any prefix of $\llbracket f_o \rrbracket$ being empty indicates that f_o is not accepted by r , i.e.*

$$\forall 1 \leq i \leq k. D_{\llbracket f_o \rrbracket[1..i]}(r) = \emptyset \implies f_o \not\models \pi_c$$

Proof. By unfolding the definition of *derivative*, we have

$$D_{\llbracket f_o \rrbracket[1..i]}(r) = \{w \mid f_o[1..i] \cdot w \in r\}$$

If this derivative is empty, it means any word preceded by $f_o[1..i]$ will not be accepted by r . Hence, we have

$$f_o = f_o[1..i] \cdot f_o[i+1..k] \notin r$$

and therefore

$$f_o \not\models \pi_c$$

□

Taking one step further, we can also check the conformance through the derivative. The key idea here is that if some word $w \in L$, then concatenating this word with the empty word remains in L , i.e. $w \cdot \varepsilon \in L$. Then based on the definition of the derivative $D_w(L) = \{s \mid w \cdot s \in L\}$, ε is in this derivative.

Lemma 2. *Let C be a class with defined methods $\overline{M}$, $\pi_c = (\llbracket \cdot \rrbracket, r)$ be a protocol of C , and o an object of class C . A final history conforms to π_c if and only if the derivative of r over $\llbracket f_o \rrbracket$ contains ε , i.e.*

$$f_o \models \pi_c \Leftrightarrow \varepsilon \in D_{\llbracket f_o \rrbracket}(r)$$

Proof. By unfolding the definition of [derivative](#), we have

$$D_{\llbracket f_o \rrbracket}(r) = \{w \mid f_o \cdot w \in r\}$$

If $f_o \models \pi_c$, then $\llbracket f_o \rrbracket \in r$ and therefore $\llbracket f_o \rrbracket \cdot \varepsilon \in r$, which implies

$$\varepsilon \in D_{\llbracket f_o \rrbracket}(r)$$

If $\varepsilon \in D_{\llbracket f_o \rrbracket}(r)$, then by the unfolding above, we have

$$r \ni \llbracket f_o \rrbracket \cdot \varepsilon = f_o$$

and hence $f_o \models \pi_c$.

□

These lemmas offer a good way to incrementally determine the conformance of the history. During the semantical reduction, we can compute a derivative every time it meets a method call and then do an empty check to detect if calling this method would eliminate the possibility of protocol conformance. At the exit point like a return statement, we can go through all the computed derivatives to check if ε is in the computed languages, which is equivalent to checking the definition of protocol conformance. In the rest of this chapter, we will utilize this idea to construct the conformance check in operational semantics and typestate systems.

3.1.2 Syntax

We introduce a simple object-oriented language CHECKOL that can check the conformance of protocols. We integrate the concept of protocol conformance into the setting of the toy language using regular expressions, mapping functions, and derivatives. The syntax of CHECKOL is shown in Fig. 3.4.

A program is encompassed by two parts: a single class declaration *ClassDecl*, and a statement S to be executed, which plays the role of the main method.

The class declaration defines a class with class name C , protocol π , a list of fields $\overline{F}$ and a list of methods $\overline{M}$. The lists $\overline{F}$ and $\overline{M}$ are possible empty sequences.

P	$::=$	$ClassDecl \ S$
$ClassDecl$	$::=$	$class \ \pi \ C \ \{ \ \overline{F} \ \overline{M} \ \}$
F	$::=$	$\top f;$
M	$::=$	$def \ m() \ \{\};$
e	$::=$	$x_i \mid e.f_i$
S	$::=$	$skip \mid x_i := x_j \mid x_i := x_j.f_k \mid x_i.f_j := x_k \mid x_i.m_j() \mid return \mid S; S$
π	$::=$	$(\llbracket \cdot \rrbracket, regex)$
$\llbracket \cdot \rrbracket$	$::=$	$\{m_1 \mapsto a_1, \dots, m_n \mapsto a_n\}$
$regex$	$::=$	$regex + Term \mid Term$
$Term$	$::=$	$Term \cdot Factor \mid Factor$
$Factor$	$::=$	$Base^* \mid Base$
$Base$	$::=$	$Letter \mid (regex)$
$Letter$	$::=$	$a_1 \mid \dots \mid a_n \mid \varepsilon$
f, m	$:$	$identifier$
$a_1, \dots, a_n$	$::=$	letter a-z, A-Z, or ε
$x_1, \dots, x_n$		pre-defined variables of class C

Figure 3.4: Syntax of CHECKOL .

The field names and method names in the lists are numbered as $\overline{f} = \{f_1, f_2, \dots, f_n\}$ and $\overline{m} = \{m_1, m_2, \dots, m_k\}$.

A field is declared with a name f . The type of f is not specified; therefore, it can be assigned to any value. Another way to interpret it is that all the fields have the $\top$ type. The $\top$ type stands for any possible value; therefore can be assigned to any value as well. This setting is reasonable because we only have addresses as values that are stored on the heap.

The method is defined with a name m . The return type is assumed to return nothing; therefore, all the methods are of *void* type without specifying. The method in CHECKOL is just an event, not The method body is ignored to avoid the inconsistency between the protocol and the method implementation. We assume the method implementation is fully encoded by the protocol, i.e. the conformance of the protocol is equivalent to the desired behavior of the method implementation. We leave the consistency checking for future work.

An expression is either a variable ranging from x_1 to x_z where z is some positive integer, or a field access of $f_i \in \overline{f}$ on some other expression e . Each variable refers to some object of type C on the heap.

A statement can be a variable assignment $x_i := x_j$, a field access $x_i := x_j.f_k$, a field update $x_i.f_j := x_k$, a method call $x_i.m_j()$, or a statement composition $S; S$. All i, j, k are assumed to be within the bounds. Conditionals and loops are also not included.

A protocol π is given as annotation (with @ notation) before the class name. It contains two parts: a mapping $\llbracket \cdot \rrbracket$ that maps every method name to a letter, and a regular language specified by regular expression *regex* over the letters used in $\llbracket \cdot \rrbracket$. The syntax of *regex* is also given in the figure, where the concatenation is written explicitly as $\cdot$. We omit this operator for simplicity in the rest of this thesis as we did for the concatenation of regular expressions.

The following is an example of a file class written in CHECKOL .

Example 3. An example file class is shown in Fig. 3.5. It contains two fields: *name*, and *content*, and three methods: *open*, *write*, and *close* with corresponding letters *o*, *w*, and *c*, respectively. The protocol of the class is declared as $(ow^*c)^*$ meaning that 1) open should always be called first, 2) write can be called arbitrary times in between a pair of open and close, and 3) a close must be called after an open.

The main method does the following: assign the content of x_1 to x_2 , then through x_2 , it opens the file, writes to the file twice, and closes the file. Finally, a return statement is added at the end suggesting the termination of the main method.

```
class @({open→o,
        write→w,
        close→c},
        (ow*c)*) File{
    name;
    content;

    def open() {};
    def write() {};
    def close() {};
}

x2 = x1
x2.open();
x2.write();
x2.write();
x2.close();
return;
```

Figure 3.5: A file class with open, write and close method.

3.1.3 Operational Semantics

We give the semantics of CHECKOL along with the runtime memory model. The runtime model storing is shown in Fig. 3.6. The only value stored in memory is a reference to the objects on the heap. There is no primitive constant (e.g. integer, boolean) in the language. It is reasonable to ignore primitive constants since we don't have control flow or computational expressions in the syntax.

The program state consists four main parts: 1) the heap h that maps addresses to objects, which is a list of fields, 2) the store σ that maps variables to values, 3)

the *object history store* τ that maps the object addresses to the current history of that object, and 4) the program point pc that uniquely represent the current point of reduction. The mapping and regular expression are interpreted as a map from method names to letters or ε , and a language this regular expression refers to. $\llbracket \cdot \rrbracket$ and r are later used as parameters in the semantics for statements to check the protocol conformance.

$$\begin{aligned}
v &::= \&l \\
l &\in \mathbb{N} \\
h &::= l \mapsto (\overline{fv}) \\
fv &::= v \\
\sigma &::= x \mapsto v \\
\tau &::= l \mapsto \Sigma^* \\
\llbracket \cdot \rrbracket &: \overline{M} \mapsto \{a_1, \dots, a_n, \varepsilon\} \\
r &::= L(regex)
\end{aligned}$$

Figure 3.6: Runtime model of CHECKOL .

As shown in Fig. 3.7 and Fig. 3.8, we use small-step semantics so that we can capture the program point, which will be used later for aliasing information. The most important setting here is that we maintain the object history as meta-information and use it to check the conformance and violation of the protocol during the reduction.

To interpret the language under our semantics, we maintain a *program state* (σ, h, τ, pc) . Initially, each variable x_i is assigned to a distinct value v_i pointing to a distinct object on the heap. Every object is also initially assigned to an empty history in τ , indicating that no method has been called at the beginning. The pc is initialized to some fresh, unique symbol to represent the starting point. Especially, we write $(\sigma_{init}, h_{init}, \tau_{init}, pc_{init})$ to represent a state satisfies the conditions above; therefore it can be used as *initial state* of a program.

CHECKOL is specially designed to exclude the executions that violate the protocol. As discussed in Lemma. 1 and Lemma. 2, we can detect some violations of the protocol earlier during the reduction by taking the derivative of the protocol language over the translated history $\llbracket h_o \rrbracket$ and checking the emptiness.

To illustrate operational semantics, we use the square bracket $\mu[a \mapsto b]$ to represent a copy of μ with only the value of a changed to b , which is also called the update of the map μ . We write v_i, v_j, v_k to indicate these meta-variables are values. The pc is updated to a fresh, unique symbol pc' after every reduction, which is omitted in the later explanation. The operational semantics of expression (Fig. 3.7) are as follows:

- Rule *S-VAR* evaluates a variable x_i by returning the value stored in σ .

$$\boxed{\langle(\sigma, h, \tau, pc), e\rangle \rightsquigarrow \langle(\sigma', h', \tau', pc'), e'\rangle}$$

$$\text{(S-VAR)} \quad \frac{}{\langle(\sigma, h, \tau, pc), x_i\rangle \rightsquigarrow \langle(\sigma, h, \tau, pc'), \sigma(x_i)\rangle}$$

$$\begin{array}{c}
\text{(S-FSTEP)} \\
\frac{\langle(\sigma, h, \tau, pc), e\rangle \rightsquigarrow \langle(\sigma, h, \tau, pc'), e'\rangle}{\langle(\sigma, h, \tau, pc), e.f_i\rangle \rightsquigarrow \langle(\sigma, h, \tau, pc'), e'.f_i\rangle}
\end{array}
\quad
\begin{array}{c}
\text{(S-FEVAL)} \\
\frac{h(l)(f_i) = v}{\langle(\sigma, h, \tau, pc), \&l.f_i\rangle \rightsquigarrow \langle(\sigma, h, \tau, pc'), v\rangle}
\end{array}$$

Figure 3.7: Operational semantics of CHECKOL for expression.

- Rule *S-FSTEP* evaluates a field look up $e.f_i$ when the precedence e can be further reduced to e' . It replaces e with the stepped expression e' .
- Rule *S-FEVAL* evaluates a field look up $e.f_i$ when the precedence is an address $\&l$. It gets the field value of object at location l by computing $h(l)(f_i)$.

The operational semantics of statement (Fig. 3.8) are as follows:

- Rule *S-SKIP* reduces the *skip* statement to itself with nothing changed in the state except for the program point pc assigned to a new value pc' .
- Rule *S-ASSIGN* reduces the variable assignment $x_i := x_j$. It evaluates the value of x_j , denoted as v_j , then produces a new machine state where store σ is updated by $\sigma[x_i \mapsto v_j]$.
- Rule *S-FACC* reduce the field access $x_i := x_j.f_k$. It computes the value of $x_j.f_k$, denoted as v_k , in two steps of expression evaluation. Then it produces a new machine state where the store σ is updated by $\sigma[x_i \mapsto v_k]$.
- Rule *S-FUPDATE* reduces the field update $x_i.f_j := x_k$. It computes the value of x_i and x_k , denoted as v_i and v_k , respectively. Then it produces a new machine state where the heap is updated by $h(v_i)[f_j \mapsto v_k]$.
- Rule *S-MCALL* reduces the method call statement $x_i.m_j()$. It first evaluates the value of x_i , denoted as v_i . It then compute the new history of the object h_o by appending the new letter for the called method $\llbracket m_j \rrbracket$ to the end of current history $\tau(v_i)$. Finally, by the result of Lemma. 1, it checks if the derivative of r over the new history h_o is non-empty.
- Rule *S-SSTEP* reduces the composition statement $S_1; S_2$ if S_1 is not a skip statement. It reduces S_1 to S'_1 and creates a new remaining statement $S'_1; S_2$.
- Rule *S-SKIP* reduces the composition statement of form *skip*; S_2 . It simply removes the *skip* statement and leaves S_2 for further reduction.

$$\langle (\sigma, h, \tau, pc), S \rangle \rightsquigarrow_{\mathbb{I}, r} \langle (\sigma', h', \tau', pc'), S' \rangle$$

(S-SKIP)

$$\frac{}{\langle (\sigma, h, \tau, pc), skip \rangle \rightsquigarrow_{\mathbb{I}, r} \langle (\sigma, h, \tau, pc'), skip \rangle}$$

(S-ASSIGN)

$$\frac{\langle (\sigma, h, \tau, pc), x_j \rangle \rightsquigarrow \langle (\sigma, h, \tau, pc'), v_j \rangle}{\langle (\sigma, h, \tau, pc), x_i := x_j \rangle \rightsquigarrow_{\mathbb{I}, r} \langle (\sigma[x_i \mapsto v_j], h, \tau, pc'), skip \rangle}$$

(S-FACC)

$$\frac{\begin{array}{l} \langle (\sigma, h, \tau, pc), x_j.f_k \rangle \rightsquigarrow \langle (\sigma, h, \tau, pc''), v_j.f_k \rangle \\ \langle (\sigma, h, \tau, pc''), v_j.f_k \rangle \rightsquigarrow \langle (\sigma', h', \tau', pc'), v_k \rangle \end{array}}{\langle (\sigma, h, \tau, pc), x_i := x_j.f_k \rangle \rightsquigarrow_{\mathbb{I}, r} \langle (\sigma[x_i \mapsto v_k], h, \tau, pc'), skip \rangle}$$

(S-FUPDATE)

$$\frac{\begin{array}{l} \langle (\sigma, h, \tau, pc), x_i \rangle \rightsquigarrow \langle (\sigma, h, \tau, pc''), v_i \rangle \\ \langle (\sigma, h, \tau, pc''), x_k \rangle \rightsquigarrow \langle (\sigma', h', \tau', pc'), v_k \rangle \end{array}}{\langle (\sigma, h, \tau, pc), x_i.f_j := x_k \rangle \rightsquigarrow_{\mathbb{I}, r} \langle (\sigma, h(v_i)[f_j \mapsto v_k], \tau, pc'), skip \rangle}$$

(S-MCALL)

$$\frac{\begin{array}{l} \langle (\sigma, h, \tau, pc), x_i \rangle \rightsquigarrow \langle (\sigma, h, \tau, pc'), v_i \rangle \\ w = \tau(v_i) \cdot \llbracket m_j \rrbracket \quad D_w(r) \neq \emptyset \end{array}}{\langle (\sigma, h, \tau, pc), x_i.m_j() \rangle \rightsquigarrow_{\mathbb{I}, r} \langle (\sigma, h, \tau[v_i \mapsto w], pc'), skip \rangle}$$

(S-SSSTEP)

$$\frac{\langle (\sigma, h, \tau, pc), S_1 \rangle \rightsquigarrow_{\mathbb{I}, r} \langle (\sigma', h', \tau', pc'), S'_1 \rangle}{\langle (\sigma, h, \tau, pc), S_1; S_2 \rangle \rightsquigarrow_{\mathbb{I}, r} \langle (\sigma, h, \tau, pc'), S'_1; S_2 \rangle}$$

(S-SSKIP)

$$\frac{}{\langle (\sigma, h, \tau, pc), skip; S_2 \rangle \rightsquigarrow_{\mathbb{I}, r} \langle (\sigma, h, \tau, pc'), S_2 \rangle}$$

(S-RETURN)

$$\frac{\forall l \in Dom(h) \cdot \varepsilon \in D_{\tau(l)}(s)}{\langle (\sigma, h, \tau, pc), return \rangle \rightsquigarrow_{\mathbb{I}, r} \langle (\sigma, h, \tau, pc'), skip \rangle}$$

Figure 3.8: Operational semantics of CHECKOL for statement.

- Rule *S-RETURN* reduces exit statement *return*. For all the created object on the heap, it computes the derivative over the (final) history of the object $\tau(l)$. Then by the result of Lemma. 2, it checks if ε is in the derivative. We simply reduce *return* to *skip* without specifying termination because we have assumed that *return* is only used at the exit point of the straight-line program. Explicit termination symbol might be needed if multiple exit points exist.

We use $\rightsquigarrow^*$ and $\rightsquigarrow_{\llbracket \cdot \rrbracket, r}^*$ to represent the transitive closure of our semantics. We say a statement S is *stuck* if there exists a state and remaining statement S' it can step to, but can take no other step.

Definition 11. Given a program P with class C and statement S , a Class C 's protocol $\pi = (\llbracket \cdot \rrbracket, r)$, and an initial state $(\sigma_{init}, h_{init}, \tau_{init}, pc_{init})$, we define a predicate *stuck*(S)

$$\begin{aligned} stuck(S) := & \exists S', (\sigma_{init}, h_{init}, \tau_{init}, pc_{init}). \\ & \langle (\sigma_{init}, h_{init}, \tau_{init}, pc_{init}), S \rangle \rightsquigarrow_{\llbracket \cdot \rrbracket, r} \langle (\sigma', h', \tau', pc'), S' \rangle \wedge \\ & \neg (\exists S'', (\sigma'', h'', \tau'', pc'') \cdot \langle (\sigma', h', \tau', pc'), S' \rangle \rightsquigarrow_{\llbracket \cdot \rrbracket, r} \langle (\sigma'', h'', \tau'', pc''), S'' \rangle) \end{aligned}$$

3.2 Typestate System

We introduce the type system that soundly checks the conformance of the protocol. The system is flow-sensitive, since the abstract history store may change after every execution of the statement. It also utilizes the alias information about how the objects might be referenced. The alias information is considered as input of the type system along with the program. Usually, alias information is produced by other static analyses, such as pointer analysis, which is often expensive to perform.

In this section, we give a brief description and definition of the alias information we use as input to the typestate system in Section 3.2.1. Then in Section 3.2.2, we define the typestate system by its typing rules.

3.2.1 Alias Information

To build up the typestate system, we abstract the history of objects into the history of reference. This abstraction naturally introduces the need for alias information since the expressions and objects are not one-to-one mapped. The process to get the alias information is not within the scope of this thesis. We assume we have a pre-processing that produces trusted alias information.

Informally, we say that two expressions do not alias if they *always* refer to distinct objects; otherwise, they *may alias*. The alias information is the functions that summarize the alias relations for the expressions in the given program. For a program P , *alias information* is given by an oracle function $MayOracle(e, pc)$,

where e is an expression and pc is the program point. The return value of the oracle is a set of expressions that can be the alias of e at pc .

The alias analysis is not trivial. The expressions refer not only to the set of xs but also may contain field accesses. For example, the oracle should also be able to find the alias information of $x_i.f_j$ at pc . Since we accumulate the field accesses again and again, the space of expression becomes potentially infinite.

For any foreign input oracles, it is just a function that returns an arbitrary set of expressions. We need to introduce the soundness of the *MayOracle* to describe what are the trustworthy oracles.

Definition 12. Given a program P of statement S and a program point pc , a *May-Oracle* is sound if and only if for any program state that can be reached at pc by reducing P from some initial state and an expression e_1 , $MayOracle(e_1, pc)$ does not contain expression e_2 implies they will not points to same object.

$$\begin{aligned} \langle (\sigma, h, \tau, pc_{init}), S \rangle \rightsquigarrow_{\llbracket \cdot \rrbracket, r} \langle (\sigma, h, \tau, pc'), S' \rangle \wedge \\ e_2 \notin MayOracle(e_1, pc) \wedge \\ \langle (\sigma, h, \tau, pc), e_1 \rangle \rightsquigarrow \langle (\sigma, h, \tau, pc'), v_1 \rangle \wedge \\ \langle (\sigma, h, \tau, pc), e_2 \rangle \rightsquigarrow \langle (\sigma, h, \tau, pc'), v_2 \rangle \implies v_1 \neq v_2 \end{aligned}$$

Intuitively, given a program P , a $MayOracle(e, pc)$ is *sound* if and only if the oracle returns all expressions that may be aliased with expression e at program point pc . Another way to interpret the soundness of $MayOracle$ is that it excludes all expressions that must not be aliased with e at pc in all executions of P .

3.2.2 Typing Rules

Typestate is a reference level approximation of the actual conformance status of the referenced object [13]. Since the typestate is computed over references, we might miss method calls because of aliasing, which makes the typestate an approximation of the actual conformance status. Therefore, we utilize the alias information to improve the precision of the analysis to an acceptable level, which is adapted from [19, 50]. Also, for the possible joining of histories, we directly define typestate as the computed derivatives so that we can use the addition operation in regular expression to join different typestates of aliases.

In the typestate system, we have two abstract state contexts: an *abstract store* $\phi(e)$, where e is an expression, and the program point pc introduced in semantics. The abstract store is a mapping from expressions to the typestate of the referenced object. The protocol $\pi = (\llbracket \cdot \rrbracket, r)$ and external oracle $O = MayOracle(e, pc)$ are added as parameters of the type system. We denote such a parameterized typestate system as $TS_{O, \pi}$. The typestates for all variables are initially set to r .

The typing rules are shown in Fig. 3.9. To better distinguish the change of typestate and the use of alias information in method calls, we use two different colors in *TS-MCALL*, blue for alias information and red for typestate update.

$$\boxed{(\phi, pc) \vdash_{\pi, O} S \dashv (\phi', pc')}$$

(TS-SKIP)

$$\frac{}{(\phi, pc) \vdash_{\pi, O} \text{skip} \dashv (\phi, pc')}$$

(TS-ASSIGN)

$$\frac{\begin{array}{l} \forall e \in \Phi(x_i. *, \phi) . e' = e[x_j/x_i] \wedge \\ T'_{e'} = \phi(e') \wedge A_e^{may} = \text{MayOracle}(e, pc) \\ \phi' = \phi[\forall e \in \Phi(x_i. *, \phi), a_e \in A_e^{may} . e \mapsto T'_{e'} \wedge a_e \mapsto T'_{e'} + \phi(a_e)] \end{array}}{(\phi, pc) \vdash_{\pi, O} x_i := x_j \dashv (\phi', pc')}$$

(TS-FACC)

$$\frac{\begin{array}{l} \forall e \in \Phi(x_i. *, \phi) . e' = e[x_j.f_k/x_i] \wedge \\ T'_{e'} = \phi(e') \wedge A_e^{may} = \text{MayOracle}(e, pc) \\ \phi' = \phi[\forall e \in \Phi(x_i. *, \phi), a_e \in A_e^{may} . e \mapsto T'_{e'} \wedge a_e \mapsto T'_{e'} + \phi(a_e)] \end{array}}{(\phi, pc) \vdash_{\pi, O} x_i := x_j.f_k \dashv (\phi', pc')}$$

(TS-FUPDATE)

$$\frac{\begin{array}{l} O^{may} = \text{MayOracle}(x_i, pc) \\ \forall y \in O^{may} \cup \{x_i\}, e \in \Phi(y.f_j. *, \phi) . e' = e[x_k/y.f_j] \wedge T'_{e'} = \phi(e') \wedge \\ A_e^{may} = \text{MayOracle}(e, pc) \\ \phi' = \phi[\forall e \in \Phi(x_i.f_j. *, \phi), a_e \in A_e^{may} . e \mapsto T'_{e'} \wedge a_e \mapsto T'_{e'} + \phi(a_e)] \\ [\forall y \in O^{may}, e \in \Phi(y.f_j. *, \phi), a_e \in A_e^{may} . e \mapsto T'_{e'} + \phi(e) \wedge a_e \mapsto T'_{e'} + \phi(a_e)] \end{array}}{(\phi, pc) \vdash_{\pi, O} x_i.f_j := x_k \dashv (\phi', pc')}$$

(TS-MCALL)

$$\frac{\begin{array}{l} T = \phi(x_i) \quad T' = D_{[m_j]}(T) \quad T' \neq \emptyset \quad A^{may} = \text{MayOracle}(x_i, pc) \\ \phi' = \phi[x_i \mapsto T'][\forall a \in A^{may} . a \mapsto T' + \phi(a)] \end{array}}{(\phi, pc) \vdash_{\pi, O} x_i.m_j() \dashv (\phi', pc')}$$

(TS-SEQ)

$$\frac{(\phi, pc) \vdash_{\pi, O} S_1 \dashv (\phi'', pc'') \quad (\phi'', pc'') \vdash_{\pi, O} S_2 \dashv (\phi', pc')}{(\phi, pc) \vdash_{\pi, O} S_1; S_2 \dashv (\phi', pc')}$$

(TS-RETURN)

$$\frac{\forall e \in \text{Dom}(\phi) . \varepsilon \in \phi(e)}{(\phi, pc) \vdash_{\pi, O} \text{return} \dashv (\phi, pc)}$$

Figure 3.9: Typing rules of the typestate system over the language.

We define an additional notation $\Phi(e. *, \phi)$ to represent all expressions in the domain of ϕ starting with e . We use $e[u/v]$ to represent substitution in e that replaces variable v by variable u . We write $\phi[\forall e \in A . e \mapsto S(e)]$ to represent a batch update in ϕ that maps every expression e in A to a typestate derived from e . We also use this qualified formula in the premises to show that the condition holds for every qualified variable.

- Rule *TS-SKIP* types the skip statement with nothing changed in abstract store and pc changed to a new value pc' .
- Rule *TS-ASSIGN* types the variable assignment $x_i := x_j$. It produces a new abstract store by updating the typestate of x_i and its aliases. The typestate of themselves and their fields need to be updated. We find the expressions of the variables and their fields by helper function Φ . Then we do a substitution to replace the variable with x_j and get the typestates $T'_{e'}$ of the new expressions. We update ϕ with $T'_{e'}$ for x_i . Since the assignment redirect all the expressions starting with x_i , we also update ϕ with $T'_{e'} + \phi(e)$ for aliases of those expressions. Finally, the pc is incremented, which is the same for every statement and is therefore omitted in the following explanation.
- Rule *TS-FACC* types the field access $x_i := x_j.f_k$. It produces a new abstract store by updating the typestate of x_i and all its aliases. The update process is the same as the previous rule except that we replace x_i and its aliases by $x_j.f_k$ instead.
- Rule *TS-FUPDATE* types the field update $x_i.f_j := x_k$. It finds all the may-alias y of x_i including x_i itself. Then for all the expression $e \in \Phi(y.f_j, *, \phi)$ that starts with $y.f_j$, let $e' = e[x_k/y.f_j]$ and $T'_{e'} = \phi(e')$, do the following things:
 1. It updates ϕ with e mapping to $T'_{e'}$.
 2. It queries the MayOracle to get the set of May-aliases of e , denoted as A_e^{may} . Then for each alias $a_e \in A_e^{may}$, it updates ϕ with a_e mapping to $T'_{e'} + \phi(b_n)$.
- Rule *TS-MCALL* types the method call $x_i.m_j()$. It first fetches the typestate of the receiver before the method is called, which is $S = \phi(x_i)$. Then it computes the new typestate after the method call by taking the derivative of S over the corresponding letter $\llbracket m_j \rrbracket$, denoted by $T' = D_{\llbracket m_j \rrbracket}(S)$. And if T' is not empty, the rule does following two updates:
 1. It updates ϕ with x_i mapping to T' .
 2. It first queries MayOracle to get the set of May-aliases of x_i , denoted as A^{may} . Then for every alias $a \in A^{may}$, The new typestate T' is merged to the existing typestate because this alias a may alias to x_i . This merging is done by updating ϕ with a mapping to $T' + \phi(a)$.
- Rule *TS-SEQ* types the sequential statement $S_1; S_2$. It divides the statements into two parts S_1 and S_2 . Then type S_1 under current abstract store ϕ resulting in a new abstract store ϕ'' . Then type S_2 under this new ϕ'' resulting in the final abstract store ϕ' .
- Rule *TS-RETURN* types the final termination statement *return*. It checks for all expressions e whose typestate is stored in ϕ , its typestate $\phi(e)$ must contain ϵ . This has the same idea as the final return in the operational semantics, the only difference is that it checks for all the expressions stored, not just the variables.

We give some examples to further illustrate the first four rules.

Example 4. Given some intermediate typing context (ϕ, pc) where ϕ contains the mapping

$$\{x_1 \mapsto aba^*, x_1.f \mapsto aba^*, x_2 \mapsto a^*, x_2.f \mapsto ba^*, x_3 \mapsto aba^*, x_3.f = aba^*\}$$

and an Oracle such that $MayOracle(x_1, pc) = x_3$ and $MayOracle(x_1.f, pc) = x_3.f$. Additionally, we have two methods a and b from class definition such that $\llbracket a \rrbracket = a$ and $\llbracket b \rrbracket = b$. Now we give examples of the typing results of all non-composition statements under this setting.

- **TS – ASSIGN** : The typing check of the statement $x_1 := x_2$ will have the form $(\phi, pc) \vdash_{\pi, O} x_1 := x_2 \dashv (\phi', pc')$, resulting in the new abstract store ϕ'

$$\begin{aligned} \phi' &= \phi[\forall e \in \{x_1, x_1.f\}, a_e \in A_e^{may}. \\ &\quad e \mapsto \phi(e[x_2/x_1]) \wedge a_e \mapsto \phi(e[x_2/x_1]) + \phi(a_e)] \\ &= \phi[x_1 \mapsto \phi(x_2), x_3 \mapsto \phi(x_2) + \phi(x_3), \\ &\quad x_1.f \mapsto \phi(x_2.f), x_3.f \mapsto \phi(x_2.f) + \phi(x_3)] \end{aligned}$$

We apply these updates and get the new mapping

$$\{x_1 \mapsto a^*, x_1.f \mapsto ba^*, x_2 \mapsto a^*, x_2.f \mapsto ba^*, x_3 \mapsto a^* + aba^*, x_3.f = ba^* + aba^*\}$$

- **TS – FACC** : We first further assume that $\phi(x_2.f.f) = a^*$. Then the typing check of the statement $x_1 := x_2.f$ will have the form $(\phi, pc) \vdash_{\pi, O} x_1 := x_2 \dashv (\phi', pc')$, resulting in the new abstract store ϕ'

$$\begin{aligned} \phi' &= \phi[\forall e \in \{x_1, x_1.f\}, a_e \in A_e^{may}. \\ &\quad e \mapsto \phi(e[x_2.f/x_1]) \wedge a_e \mapsto \phi(e[x_2/x_1]) + \phi(a_e)] \\ &= \phi[x_1 \mapsto \phi(x_2), x_3 \mapsto \phi(x_2) + \phi(x_3), \\ &\quad x_1.f \mapsto \phi(x_2.f.f), x_3.f \mapsto \phi(x_2.f.f) + \phi(x_3)] \end{aligned}$$

We apply these updates and get the new mapping

$$\{x_1 \mapsto ba^*, x_1.f \mapsto a^*, x_2 \mapsto a^*, x_2.f \mapsto ba^*, x_3 \mapsto ba^* + aba^*, x_3.f = a^* + aba^*\}$$

- **TS – FUPDATE** : We first further assume that $MayOracle(x_1.f, pc) = x_4$ and $\phi(x_4) = aba^*$. Then the typing check of the statement $x_1.f := x_2$ will have the form $(\phi, pc) \vdash_{\pi, O} x_1.f := x_2 \dashv (\phi', pc')$, resulting in the new abstract store ϕ'

$$\begin{aligned} \phi' &= \phi[\forall e \in \{x_1.f\}, a_e \in A_e^{may}. e \mapsto \phi(e[x_2/x_1.f]) \wedge a_e \mapsto \Phi(a.*, \phi)] \\ &\quad [\forall y \in \{x_3\} e \in \Phi(y.f.*, \phi), a_e \in A_e^{may}. \\ &\quad e \mapsto \phi(e[x_2/y.f]) + \phi(e) \wedge a_e \mapsto \phi(e[x_2/y.f]) + \phi(e)] \\ &= \phi[x_1.f \mapsto \phi(x_2), x_4 \mapsto \phi(x_2) + \phi(x_4)] \\ &\quad [x_3.f \mapsto \phi(x_2.f) + \phi(x_3)] \end{aligned}$$

We apply these updates and get the new mapping

$$\begin{aligned} \{x_1 \mapsto aba^*, x_1.f \mapsto a^*, x_2 \mapsto a^*, x_2.f \mapsto ba^*, \\ x_3 \mapsto aba^*, x_3.f = ba^* + aba^*, x_4 = a^* + aba^*\} \end{aligned}$$

- **TS – MCALL** : The typing check of the statement $x_1 := x_2$ will have the form $(\phi, pc) \vdash_{\pi, O} x_1.a() \dashv (\phi', pc')$, resulting in the new abstract store ϕ'

$$\begin{aligned}\phi' &= \phi[x_1 \mapsto D_a(\phi(x_1))] \\ &\quad [\forall a \in \{x_3\} . a \mapsto D_a(\phi(x_1)) + \phi(a)] \\ &= \phi[x_1 \mapsto D_a(aba^*), x_3 \mapsto D_a(aba^*) + \phi(x_3)]\end{aligned}$$

We apply these updates and get the new mapping

$$\{x_1 \mapsto ba^*, x_1.f \mapsto aba^*, x_2 \mapsto a^*, x_2.f \mapsto ba^*, x_3 \mapsto ba^* + aba^*, x_3.f = aba^*\}$$

Finally, with all the typing rules defined above, we also define the soundness of this typestate system with respect to the operational semantics defined in Fig. 3.1.3.

Definition 13. A typestate system $TS_{O, \pi}$ with oracle O and protocol $\pi = (\llbracket \cdot \rrbracket, r)$ is *sound* if and only if any well-typed statement will not become stuck at runtime in any execution with respect to the operational semantics defined in Fig. 3.1.3. Formally, we say

$$(\phi_{init}, pc_{init}) \vdash_{\pi, O} S \dashv (\phi', pc') \implies \langle (\sigma, h, \tau, pc_{init}), S \rangle \rightsquigarrow_{\llbracket \cdot \rrbracket, r} \langle (\sigma', h', \tau', pc'), T' \rangle$$

where (σ, h, τ) is a valid initial state.

Intuitively, the soundness of the typestate system $TS_{O, \pi}$ means that it will report all the statements that may get stuck or violate the protocol. In other words, the typestate system is sound if and only if it only accepts the statements that will never get stuck or violate π .

Chapter 4

Pumping Closure

In this chapter, we explore a new family of regular languages. Section 4.1 defines *Pumping word* and the closure under this concept. We then show the other representation of the closure in regular expression and DFA. We also look into the properties of the closure. In Section 4.2, *interleaving shuffle* of two given words are defined. We then further discover two special groups of pumping closure by restricting the base language. We finally show that these two special groups of pumping closure is closed under interleaving shuffle.

4.1 Pumping Closure

We start with a special relation that extends a given word by repeating a letter in the word.

Definition 14 (Pumping Word). A word $s' \in \Sigma^*$ is a *pumping word* of $s \in \Sigma^*$, if $s' = s[1..k] \cdot s[i] \cdot s[k+1..|s|]$, where $1 \leq i \leq k \leq |s|$.

Intuitively, a pumping word is produced by pumping a letter in the given word somewhere after its occurrence. Here is a simple example.

Example 5. Consider the word $s = abc$, one possible pumping word of s is $s' = ab \cdot a \cdot c$. This is because it can also be represented as $s' = s[1..2] \cdot s[1] \cdot s[3..3]$. Moreover, below are all the possible pumping words of s

$$a \cdot a \cdot bc, \quad ab \cdot a \cdot c, ab \cdot b \cdot c, \quad abc \cdot a, abc \cdot b, abc \cdot c$$

Def. 14 can be applied to determine whether a given word is a pumping word of another given word. However, the example above also shows that it is possible to generate pumping words from a given word. We summarize this generation in three steps:

1. First, we select a letter from the word s by finding an index $i \in [1, |s|]$.

2. Second, we find another index later to insert this letter, i.e. find $k \in [i, |s|]$.
3. Finally, construct the pumping word $s' = s[1..k] \cdot s[i] \cdot s[k+1..|s|]$.

There are two constraints to this pumping. First, a letter can be pumped into the word only if it appears in the word. Second, a letter can be pumped only after the first occurrence of the same letter. The procedure above shows that the generation has more than one choice as long as $|s| > 1$. The number of pumping words is also bound by $|s|^2$. We write

$$\text{Pump}(s) = \{s' \mid s' \text{ is a pumping word of } s\}$$

for all the possible pumping words of a given word s . Additionally, we define the set of language after more than one pumping operation.

Definition 15. Given a language $s \in \Sigma^*$, we recursively define the pumping word as

- $\text{Pump}^0(s) = s$
- $\text{Pump}^{n+1}(s) = \text{Pump}(\text{Pump}^n(s)) = \sum_{w \in \text{Pump}^n(s)} \text{Pump}(w)$

It is well defined because the size of $\text{Pump}(s)$ is bound; therefore, it is always finite for any $\text{Pump}^n(s)$. We can take the pumping words of all the words from $\text{Pump}(s)$. The result language is also bound; hence, finite. This finiteness guarantees the union operation to be well defined.

We define the closure of a given language under the pumping word operation.

Definition 16 (Pumping Closure). Given a language $L \subseteq \Sigma^*$, a *pumping closure* of L , written as $L^\diamond$, is the smallest language such that

1. $L \subseteq L^\diamond$, and
2. $L^\diamond$ is pumping word closed, i.e. $\forall s \in L^\diamond. \text{Pump}(s) \subseteq L^\diamond$.

We first show some properties of pumping closure that are immediately led to by the inductive definition of pumping closure. The proof is trivial and therefore omitted.

Proposition 5. Let $L \subseteq \Sigma^*$, then

$$\forall k \in \mathbb{N}, s \in L. \text{Pump}^k(s) \subseteq L^\diamond$$

Lemma 3. Let $L \subseteq \Sigma^*$, then given any word $w \in L^\diamond$,

$$\exists n \in \mathbb{N}, s \in L. w \in \text{Pump}^n(s)$$

Prop. 16 and Lemma. 3 give a clearer shape of the pumping closure. In the pumping closure of a word $s^\diamond$, every word $w \in s^\diamond$ can be generated by applying the pumping word finitely many times to the base word s .

Proposition 6. Let $L \subseteq \Sigma^*$, then

$$L^\diamond = (L^\diamond)^\diamond$$

Prop. 6 shows that the operation of taking the pumping closure is idempotent, which is also a direct result from the inductive definition. The proof is trivial by applying the minimality and closedness of the closure; therefore it is omitted.

In addition to the structural discussion at the language level, we also discuss the structure of pumping closure in alternative forms. We now restrict the base language L to be regular, and show that the pumping closure of a regular language remains regular, i.e. the pumping closure operation preserves regularity. We show this by identifying a regular expression that is equal to the pumping closure [34]. This gives us more expressive ways to check and construct the pumping closure.

We start with the regularity of the pumping words of a single word. The regularity is trivial because of the finiteness of the language. We also give the following regular expression form.

Proposition 7. Given $s = a_1 \cdots a_n \in \Sigma^*$, we have

$$\begin{aligned} Pump(s) &= \sum_{i=1}^n a_1 \cdots a_i \left(\sum_{j=1}^i a_j \right) a_{i+1} \cdots a_n \\ &= a_1 \cdot (a_1) \cdot a_2 \cdots a_n + \\ &\quad a_1 a_2 \cdot (a_1 + a_2) \cdot a_3 \cdots a_n + \\ &\quad \cdots + \\ &\quad a_1 \cdots a_n \cdot (a_1 + \cdots + a_n) \end{aligned}$$

The proof of this proposition is straightforward, since it just lists all the pumping options after each letter and adds them together.

We consider the regularity of pumping closure. The regularity is proved by constructing its regular expression. We state the theorem first and present its proof later.

Theorem 4. *Regular language is closed under pumping word, i.e. the pumping closure of a regular language is still regular.*

We divide the proof into three main parts: First, we prove the regularity of the pumping closure of a single word by constructing a regular expression that represents the same language. Second, we prove a lemma that states that the (at least finite) summation and pumping closure operations distribute over union. i.e. the union of two pumping closures is the pumping closure of the union of two base languages. Finally, we show that the pumping closure of the language represented by regular expressions can be reduced into a finite summation of pumping closures of a single word.

We first give the regular expression of pumping closure of a single word as follows and prove the bidirectional inclusion between this expression and the pumping closure.

Lemma 5. Given a word $s = a_1 \cdots a_n \in \Sigma^*$,

$$s^\diamond = \prod_{i=1}^n \left[a_i \left(\sum_{j=1}^i a_j \right)^* \right] = a_1 a_1^* \cdots a_n (a_1 + \cdots + a_n)^*$$

Proof. For convenience, we denote $R = \prod_{i=1}^n \left[a_i \left(\sum_{j=1}^i a_j \right)^* \right]$. We show

$$s^\diamond \subseteq R \wedge R \subseteq s^\diamond$$

For any word $w \in R$, by the definition of regular expression, w can always be split into a form of

$$w = \prod_{k=1}^n (a_k w_k) = a_1 w_1 a_2 w_2 \cdots a_n w_n$$

where $w_i \in (\sum_{j=1}^i a_j)^*$ for $i \in [1, n]$.

We demonstrate $s^\diamond \subseteq R$ by establishing that R is closed under the pumping operation, i.e., $\text{Pump}(w) \subseteq R$. Note that a single letter will be added to w , either between two letters a_i and a_{i+1} or at the end of the word after a_n . Consequently, the pumping word of w is expressed similarly as w , with only one altered w_i . Let w'_k denote the modified subword for some $k \in [1, n]$. By the pumping word's definition, we also know that $w'_i \in A_j^*$ and hence $w' \in R$ and $\text{Pump}(w) \subseteq R$. Consequently, $s^\diamond \subseteq R$ holds due to the minimality minimal nature of $s^\diamond$ for being closed under the pumping operation.

To conclude, we demonstrate that R is contained within $s^\diamond$. Assume $w = \prod_{k=1}^n a_k w_k \in R$. For any w_i where $i \in [1, n]$, define $w' = (\prod_{k=1}^{i-1} a_k w_k) \cdot a_i \cdot \prod_{k=i+1}^n a_k w_k$, which represents the word obtained by omitting w_i from w . Observe that the prefix $(\prod_{k=1}^{i-1} a_k w_k) \cdot a_i$ encompasses all letters in A_i . This implies that w_i can be reconstructed from w' by performing the pumping operation $|w_i|$ times, i.e.

$$w \in \text{Pump}^{|w_i|}(w')$$

Let $N = \sum_{i=1}^n |w_i|$. Note that this procedure is applicable to any w_i . Therefore, coupled with Prop. 5, we infer

$$w \in \text{Pump}^N(s) \subseteq s^\diamond$$

□

The following lemma shows that the pumping closure distributes over union.

Lemma 6. Given two languages $L_1, L_2 \in \Sigma^*$, we have

$$L_1^\diamond + L_2^\diamond = (L_1 + L_2)^\diamond$$

Proof. We prove this by showing bidirection inclusion. It's trivial that both $L_1^\diamond \subseteq (L_1 + L_2)^\diamond$ and $L_2^\diamond \subseteq (L_1 + L_2)^\diamond$ by definition. So we just need to show that $(L_1 + L_2)^\diamond \subseteq L_1^\diamond + L_2^\diamond$.

Let $s \in (L_1 + L_2)^\diamond$. Then by Prop. 3, we have

$$\exists n \in \mathbb{N} . s \in \text{Pump}^n(L_1 + L_2) = \text{Pump}^n(L_1) + \text{Pump}^n(L_2)$$

Then from Prop. 5, we have

$$\exists n \in \mathbb{N} . s \in \text{Pump}^n(L_1) + \text{Pump}^n(L_2) \subseteq L_1^\diamond + L_2^\diamond$$

This concludes the proof. $\square$

The lemma below shows that the pumping closure subdistributes over concatenation.

Lemma 7. *Given two languages $L_1, L_2 \in \Sigma^*$, we have*

$$L_1^\diamond \cdot L_2^\diamond \subseteq (L_1 \cdot L_2)^\diamond$$

Proof. The proof is straightforward. For any given $s = s_1 s_2 \in L_1^\diamond \cdot L_2^\diamond$, there exists $m, n \in \mathbb{N}$ such that $s_1 \in \text{Pump}^m(L_1)$ and $s_2 \in \text{Pump}^n(L_2)$. The word s can be constructed in $\text{Pump}^{m+n}(L_1 \cdot L_2)$ by doing the same pumping operations to get s_1 and s_2 . Hence $s \in \text{Pump}^{m+n}(L_1 \cdot L_2) \subseteq (L_1 \cdot L_2)^\diamond$ $\square$

Finally, we introduce the following lemma to describe the reduction of the pumping closure of a language containing the Kleene star.

Lemma 8. *Given a regular expression r , there exists a regular expression r_f such that r_f represents a finite sub-language of r 's, but the pumping closure of r_f is a superset of the pumping closure of r , i.e. $L(r_f) \subseteq L(r)$, and $r^\diamond \subseteq r_f^\diamond$*

Proof. We prove by induction on the regular expression r . There are three base cases according to Def. 5, which are empty set, empty string and a single letter. They are all trivial since they themselves represent a finite language. Setting r_f to be $\emptyset$, ε , and a prove the lemma.

There are three inductive cases, which are union, concatenation and Kleene star.

Union : For the *union* case $r = r_1 + r_2$, we have the inductive hypothesis that r_1 and r_2 satisfies the lemma. Let $r_{f,1}$ and $r_{f,2}$ be the corresponding finite regular expressions for r_1 and r_2 . By definition, we have $r_{f,1} + r_{f,2} \subseteq r_1 + r_2$. Combined with Lemma 6, we have

$$r^\diamond = (r_1 + r_2)^\diamond = r_1^\diamond + r_2^\diamond = r_{f,1}^\diamond + r_{f,2}^\diamond = (r_{f,1} + r_{f,2})^\diamond$$

Note that $r_{f,1} + r_{f,2}$ will also be finite, which can be used as r_f for r .

Concatenation : For concatenation case $r = r_1 r_2$, we still let the sets be $r_{f,1}$ and $r_{f,2}$. Similarly, we will get $r_{f,1} r_{f,2} \subseteq r_1 r_2$. We now prove $(r_1 r_2)^\diamond \subseteq (r_{f,1} r_{f,2})^\diamond$. Note

that from $r_1^\diamond = (r_{f,1})^\diamond$, we have $r_1 \subseteq r_{f,1}^\diamond$ since $r_1 \subseteq r_1^\diamond$. The same result applies to r_2 . By definition we know that concatenation preserves subset relation. We therefore have $r_1 r_2 \subseteq r_{f,1}^\diamond r_{f,2}^\diamond$. By Lemma 7, we also get

$$r_1 r_2 \subseteq r_{f,1}^\diamond r_{f,2}^\diamond \subseteq (r_{f,1} r_{f,2})^\diamond$$

Note that the closure is idempotent (from Prop. 6), and the pumping closure preserves the subset relation. So we can take the pumping closure of the languages above and get

$$(r_1 r_2)^\diamond \subseteq (r_{f,1}^\diamond r_{f,2}^\diamond)^\diamond \subseteq ((r_{f,1} r_{f,2})^\diamond)^\diamond = (r_{f,1} r_{f,2})^\diamond$$

Kleene Star : For the Kleene star case r^* , we first prove the following: there exists a number $N \in \mathbb{N}$ such that

$$\forall k \in \mathbb{N}^+ . (r^{N+k})^\diamond \subseteq (r^N)^\diamond \quad (4.1)$$

By inductive hypothesis, we have a finite language $r_f \subseteq r$ such that $r_f^\diamond = r^\diamond$. We set $N = |r_f|$ since r_f is finite. Note that r_f is finite, therefore for any positive integer $k \in \mathbb{N}^+$, r_f^{N+k} must repeat one word from r_f at least k times, i.e. any $s \in r_f^{N+k}$ can be represented as $s = s_1 \cdots s_{N+k}$ where $s_1, \dots, s_{N+k} \in r_f$, and there exists at least $k+1$ entries of same sub-word, i.e.

$$\exists I = \{i_1, \dots, i_{k+1} \in [1, N+k]\}, \forall i, j \in I . s_i = s_j$$

Notice all those repetitions can also be generated by pumping operation over the first word $s_{\min(I)}$. Hence we have $s \in (r_f^N)^\diamond$ and hence $r_f^{N+k} \subseteq (r_f^N)^\diamond$. Note that the Equation 4.1 is guaranteed to be well defined because the Kleene star is well defined, and the infinite summation preserves subset relation. Then since k is arbitrary, we know that Equation 4.1 holds.

Now we just select the new r'_f for r^* to be $r'_f = \sum_{i=0}^N r_f^i$. It's worth noting that the size of r'_f is $N^2 = |r_f|^2$, which is exponential. The exponential exploration is fine for a existential proof but may not be satisfactory in practical.

Then we prove the correctness of r'_f by contradiction. Suppose

Let $L_N = \{w \in r^i \mid 0 \leq i \leq N\}$ and $L_{inf} = \{w \in r^i \mid i > N\}$. We can represent $L(r^*)$ as $L(r^*) = L_N + L_\infty$ and prove $r^\diamond \subseteq (r'_f)^\diamond$. For the first part, we have

$$(r^i)^\diamond = (r r^{i-1})^\diamond \subseteq (r_f r^{i-1})^\diamond \subseteq \dots \subseteq (r_f^i)^\diamond$$

which is proven in the same manner as the concatenation case. Therefore we have

$$L_N^\diamond = \left(\sum_{i=0}^N L(r^i) \right)^\diamond = \sum_{i=0}^N (L(r^i))^\diamond \subseteq \sum_{i=0}^N (L(r_f^i))^\diamond = \left(L \left(\sum_{i=0}^N r_f^i \right) \right)^\diamond$$

We now prove the second case. Suppose there is a $w \in L_{inf}^\diamond$. Then by Lemma 3, there is a number $m \in \mathbb{N}$ such that $w \in Pump^m(L_{inf})$. Moreover, by applying Prop. 3, there

exists a word $l \in L_{inf}$ such that $w \in Pump^m(l)$ by the Def. 15. Then by definition of L_{inf} , there exists a number $p \in \mathbb{N}$ such that $p > N$ and $l \in r^p$. Hence we can apply Equation 4.1 and get

$$w \in Pump^m(l) \subseteq Pump^m(r^p) \subseteq (r^p)^\diamond \subseteq (r_f^N)^\diamond$$

Hence, we have $L_{inf} \subseteq (r_f^N)^\diamond$ and therefore

$$(r^*)^\diamond = L_N^\diamond + L_{inf}^\diamond \subseteq \left(\sum_{i=0}^n r_f^i\right)^\diamond + (r_f^N)^\diamond = (r_f^N)^\diamond$$

Note that we also have $r_f' \subseteq r^*$, therefore the subset relation is actually equality. $\square$

Now with Lemma 5-8, we have found a way to construct the regular expression for arbitrary language by taking the sum of every word pumping closure. This leads to many different forms of regular expressions. We give an example below to give a more intuitive view.

Example 6. Consider the following regular expression $r = (a^* + bc)^*$. We can build the regular expression for $r^\diamond$ in a bottom-up way.

We first construct the regular expression for $r_1 = a^*$. Since a is already a finite language and alphabet $\Sigma_1 = \{a\}$, we get $r_f^{in} = \sum_{i=0}^1 a^i = \varepsilon + a$.

Then for the outer Kleene star, the alphabet is $\Sigma_2 = \{a, b, c\}$ and the finite language $r_f^{out} = \varepsilon + a + bc$. Then we get the final $r_f' = \sum_{i=0}^3 r_f^{out}$. To make it more readable, we remove all the words with repetition and get

$$r_f = \varepsilon + a + bc + abc + bca$$

then we will get the final regular expression to be

$$r^\diamond = \varepsilon + a^* + bb^*c(b+c)^* + aa^*b(a+b)^*c(a+b+c)^* + bb^*c(b+c)^*a(a+b+c)^*$$

So far, we have connected the pumping closure to regular expression showing the regularity is preserved by taking pumping closure. We show the regularity by finding out the regular expression for the pumping closure. To better illustrate the shape of pumping closures, we put a DFA of the pumping closure of the example above in Fig. 4.1.

The DFA of single pumping closure has a similar form to the Kleene star since each Kleene star can be seen as a self-loop state and the intermediate letter is an edge from the previous star to the next star.

The pumping closure can also be seen as the union of some weakly defined equivalence classes. We first explain the equivalence class by introducing a canonical form and the equivalence relation.

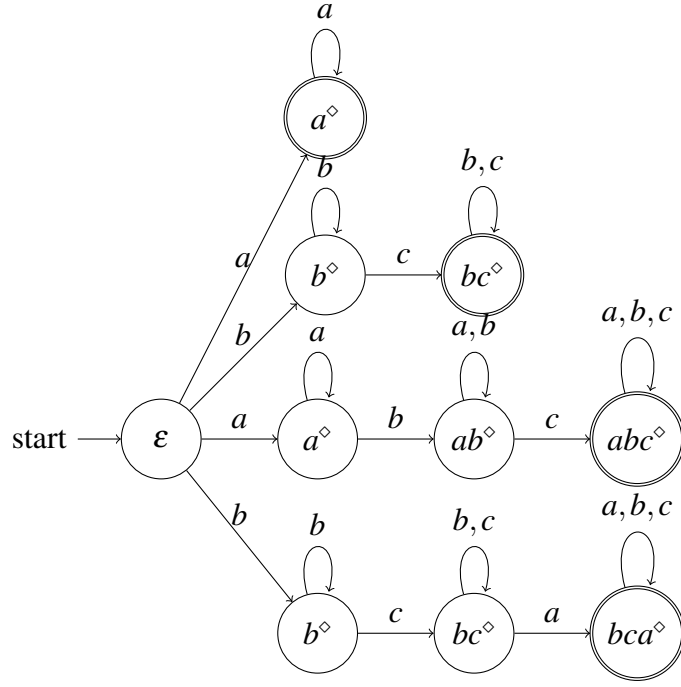


Figure 4.1: DFA of example 6

Definition 17 (Occurrence Word). A word $s \in \Sigma^*$ is an *occurrence word* if $\forall 1 \leq i, j < |s| . s[i] = s[j] \implies i = j$.

Intuitively, the occurrence word is the word that each letter from the alphabet only appears at most once. This is the starting point to generate pumping words. In other words, this is the base or canonical form we seek in the pumping words. To make use of this occurrence word, let's look at the reverse operation of the pumping operation.

Definition 18 (Reverse Pumping Word). A word $s \in \Sigma^*$ is a *reverse pumping word* of $s' \in \Sigma^*$ if either of the following is true

1. $s' = s$ is occurrence word.
2. $\exists 1 \leq i < |s| . s = s'[1..i-1]s'[i+1..|s|]$ and $s'[i] \in s'[1..i-1]$.

Intuitively, the reverse pumping word is the reverse operation of generating the pumping word, i.e. removing the repeated word. Therefore we use $Pump^{-1}(s')$ to represent all the possible reverse pumping words of s' . But here we don't construct the closure under this operation. Instead, let's consider the fixpoint of this operation.

Lemma 9. For any $s \in \Sigma^*$, there exists $n \in \mathbb{N}$ and an occurrence word $o \in \Sigma^*$ such that $Pump^{-n}(s) = Pump^{-n-1}(s) = \{o\}$.

The proof is based on the monotonicity of the number of letters in s and the equivalence formula below.

$$(\forall 1 \leq i, j < |s| . s[i] = s[j] \implies i = j) \Leftrightarrow (\forall 1 \leq i < |s| . s[i] \notin s[1..i-1])$$

The lemma above ensures that each word has a finite, unique fixpoint which is also an occurrence word. For convenience, we uniformly denote this fixpoint by $Pump^{-\infty}(s)$. Therefore, we define a relation as follows

Definition 19 (Occurrence Relation). Given $s_1, s_2 \in \Sigma^*$, $s_1 \sim s_2$ iff $Pump^{-\infty}(s_1) = Pump^{-\infty}(s_2)$.

Since the lemma above shows the reverse pumping operation for each word has a unique fixpoint, the relation above is an equivalence relation. Therefore we define equivalence classes. Additionally, we also need a weaker defined equivalence class because the pumping closure doesn't restrict the base language L to only contain occurrence words.

Definition 20 ((Weak) Equivalence Class). Given $s \in \Sigma^*$, the *equivalence class* of s w.r.t. the occurrence relation is defined as $[s] = \{l \in \Sigma^* \mid l \sim s\}$. A language L is called *weak equivalence class* of s w.r.t. the occurrence relation iff $L \subseteq [s]$.

To this point, we are able to explain why we say the pumping closure is the union of weak equivalence classes. For each word $s \in L$, the pumping closure $s^\diamond$ is a weak equivalence class of s . And it's easy to prove that the pumping closure $L^\diamond$ is the sum of all the pumping closures of words in it. Hence $L^\diamond$ is also the union of weak equivalence classes.

4.2 Special Pumping Closures

In this section, we focus on two special group of pumping closures with the base language meeting some extra constraints. The main properties of these two groups are that they are closed under some new operation. So, before introducing those extra constraints, we first introduce the new operation.

Definition 21 (Interleaving Shuffle). Given two words $s_1, s_2 \in \Sigma^*$, an *interleaving shuffle* of s_1 and s_2 is a word s such that

$$\begin{aligned} s &= r_0 l_0 \cdots r_n l_n \\ \text{where } s_1 &= r_0 \cdots r_n, \\ s_2 &= l_0 \cdots l_n, \text{ and} \\ r_0, l_0 &\text{ can be } \varepsilon \end{aligned}$$

Intuitively, interleaving takes two words as input and then combines them into one word in such a way that the characters from each original word appear in the same order they did in their respective words. For convenience, we denote all the possible interleaving shuffles of s_1 and s_2 as $s_1 \sqcup s_2$.

4.2.1 Prefix-include Pumping Closure

At first, we revisit the definition of prefix.

Definition 22 (Prefix). Given a word $s \in \Sigma^*$, a word $l \in \Sigma$ is called a prefix s iff

$$\exists w \in \Sigma^* . s = lw$$

By definition, the prefixes of a given word s can be categorized into three different types. They are 1) ε for empty word where $w = s$, 2) the whole origin word s where $w = \varepsilon$ and 3) any sub-sequence starting from the first letter $s[1..k]$ where $k = 1, \dots, |s| - 1, w = s[k + 1, |s|]$. We conclude this into a proposition.

Proposition 8. Given a word $s \in \Sigma^*$, we have

$$\text{pref}(s) = \{\varepsilon\} + \{s[1..k] \cdot k = 1, \dots, |s|\}$$

Now we add extra constraints about prefix to our pumping closure and construct some new subgroup of pumping closure.

Definition 23 (Prefix-include Pumping Closure). Given a language L , its pumping closure $L^\diamond$ is called *prefix-include* pumping closure if and only if L satisfies

$$\exists s \in L, \forall s' \in L . s' \in \text{pref}(s)$$

Intuitively, the prefix-include pumping closure restricts the base language L in the way that L contains a word s such that L only contains the prefix of s . The most straightforward case of this property is that $L = \text{pref}(s)$.

We put the main theorem for prefix-include pumping closure at the beginning before proving it with a few lemmas introduced.

Theorem 10. *The prefix-include pumping closure is closed under interleaving shuffle, i.e.*

$$\forall L \subseteq \Sigma^* . (\exists s \in L, \forall s' \in L . s' \in \text{pref}(s)) \implies (\forall s_1, s_2 \in L^\diamond . s_1 \sqcup s_2 \subseteq L^\diamond)$$

The first lemma we need is a general property for pumping closure. This decomposed the proof of the pumping relation between two words into smaller chunks of the words.

Lemma 11. *Given a language $s^\diamond$ and a partition $s = s_1s_2$, then for any other words $s'_1 \in s_1^\diamond$ and $s'_2 \in s_2^\diamond$, we have $s'_1s'_2 \in (s_1s_2)^\diamond$.*

Proof. The proof is trivial based on the fact that if a word s' is the pumping word of another word s , then adding letters at front or in the end on both words do not change this relation. This is a straight result from the definition of pumping word, i.e.

$$\forall l \in \Sigma^* . (s' \in \text{Pump}(s) \wedge l' \in \text{Pump}(l)) \implies s'l' \in \text{Pump}(s)\text{Pump}(l) \subseteq \text{Pump}(sl)$$

Then we generalize it to the pumping closure level since s is arbitrary, i.e.

$$\forall l \in \Sigma^* . (s' \in s^\diamond \wedge l' \in l^\diamond) \implies s'l' \in s^\diamond l^\diamond \subseteq (sl)^\diamond$$

□

The second lemma is a direct result by prefix-include property and Lemma 3.

Lemma 12. *Given a prefix-include pumping closure $L^\diamond$ and $s_1, s_2 \in L^\diamond$, there exists two words $s_1^o, s_2^o \in L$ such that $s_1 \in (s_1^o)^\diamond$, $s_2 \in (s_2^o)^\diamond$, and either $s_1^o \in \text{pref}(s_2^o)$ or $s_2^o \in \text{pref}(s_1^o)$.*

The third lemma connects the prefix to the pumping closure.

Lemma 13. *Given a word $s \in \Sigma^*$ and a word from its pumping closure $s' = l_0 \cdots l_n \in s^\diamond$ where $l_0, \dots, l_n$ are sub-words, we can always find a mapping from the first is sub-words from s' to a prefix of s , where*

1. *this prefix of s' is in the pumping closure of that prefix of s .*
2. *that prefix of s is the longest such prefix to fulfill last constraint.*

Formally, it can be described as

$$\begin{aligned} \exists f : [0, n] \rightarrow [1, |s|], \forall i \in [0, n]. \\ (l_0 \cdots l_i \in (s[1..f(i)])^\diamond) \wedge \\ (l_0 \cdots l_i \notin (s[1..f(i) + 1])^\diamond \vee f(i) = |s|) \end{aligned}$$

Proof. We first show that there exists such function f to fulfill the first constraint, i.e.

$$\exists f : [0, n] \rightarrow [1, |s|], \forall i \in [0, n] . (l_0 \cdots l_i \in (s[1..f(i)])^\diamond)$$

This can be proved by the fact that for any word $s = a_1 \cdots a_n$ and $s' \in s^\diamond$, s' can be described as $s' = a_1 w_1 \cdots a_n w_n$, which is used in the proof of Lemma 4. Therefore for each prefix of s' , we can find the letters from s that is used to generate this prefix.

However, the uniqueness of f is not guaranteed because we don't restrict the duplication of letters in s_1^o . For a word $w = aaa \in (aa)^\diamond$, $w[1..2] = aa$ can both be seen as $w[1..2] \in a^\diamond$ and $w[1..2] \in (aa)^\diamond$. Note the length of these prefixes are always bound, which means there must be a longest prefix satisfies this relation. □

Before we jump into the proof, we first give an example to give a rough idea of how we prove the theorem.

Example 7. Suppose we have $s = aabc \in (a + b + c)^*$. Then we have $\text{pref}(s) = \{\varepsilon, a, aa, aab, aabc\}$. We take the pumping closure of $\text{pref}(s)$, which is

$$(\text{pref}(s))^\diamond = \varepsilon + aa^* + aa^*aa^* + aa^*aa^*b(a+b)^* + aa^*aa^*b(a+b)^*c(a+b+c)^*$$

Now we take two words from the pumping closure:

$$s_1 = aaba \quad s_2 = aabcb$$

Let's look at some interleaving shuffles of these two words. We use **red** to represent the letters from s_1 , **blue** for letters from s_2 , and **purple** to represent the letters from s_3 .

$$s_1 \sqcup s_2 \ni w_1 = \text{aaabaabcb}$$

$$s_1 \sqcup s_2 \ni w_2 = \text{aabaaabab}$$

$$s_1 \sqcup s_3 \ni w_3 = \text{aaaaba}$$

The idea of the proof of Thm. 10 is to extract a prefix of s that can generate the interleaving shuffle through pumping operation. Whenever a new occurrence of letters in s is found during the traversal of the interleaving shuffle, we extract and memorize them. This process is shown to must generate a prefix of s because of the prefix property later in the proof.

For example, A prefix **aaabc**, which is actually s itself, can be extracted from w_1 . The color of the letter indicates which part of the interleaving they are extracted.

Another prefix **aaab** can also be extracted from w_2 , showing that we might extract nothing from some chunk of letters.

Finally, a prefix **aaa**, which is a prefix of s other than s itself, can be extracted from w_3 .

Finally, we give the proof for Theorem 10.

Proof. For any interleaving shuffle $s = l_0 r_0 \cdots l_n r_n \in s_1 \sqcup s_2$. We first apply Lemma 12 to get $s_1^o, s_2^o \in L$ such that $s_1 \in (s_1^o)^\diamond$ and $s_2 \in (s_2^o)^\diamond$. Without loss of generality, we suppose $s_1^o \in \text{pref}(s_2^o)$. Notice that we can also split s_1 and s_2 into

$$s_1 = l_0 \cdots l_n \quad s_2 = r_0 \cdots r_n$$

Then by applying Lemma 13, we can get two functions $f : [0, n] \rightarrow [1, |s_1^o|]$ and $g : [0, n] \rightarrow [1, |s_2^o|]$ that links the prefixes of s_1 and s_2 to prefixes of s_1^o and s_2^o .

Also, notice that in interleaving shuffle s , there is a sub-word inserted in between every (l_i, l_{i+1}) and (r_i, r_{i+1}) . To analyze this inserted sub-words, we define two maximum functions of f and h as

$$\begin{aligned} h_1(x) &:= \max(f(x), g(x)) \\ h_2(x) &:= \text{if } x == 0 \text{ then } f(0) \text{ else } \max(f(x), h_1(x-1)) \end{aligned}$$

Then we prove the following statement

$$\forall i \in [0, n] . l_0 r_0 \cdots l_i \in (s_2^o[1..h_2(i)])^\diamond \wedge l_0 r_0 \cdots l_i r_i \in (s_2^o[1..h_1(i)])^\diamond \quad (4.2)$$

We prove by induction on the number i . First we prove two base cases for $i = 0$. The first part for $i = 0$ is by definition since f is defined to satisfy $l_0 \in (s_2^o[1..f(0)])^\diamond = (s_2^o[1..h_2(0)])^\diamond$. The second part for $i = 0$ has two cases.

- First, if $f(0) \geq g(0)$, then we have

$$l_0 r_0 \in (s_2^o[1..f(0)])^\diamond = (s_2^o[1..h_1(0)])^\diamond$$

since every letter in r_0 has appeared before in l_0 .

- Second, if $f(0) < g(0)$, we do another partition $r_0 = r_0^1 r_0^2$ such that

$$r_0^1 \in (s_2^o[1..f(0)])^\diamond \wedge r_0^2 \in (s_2^o[f(0) + 1..g(0)])^\diamond$$

This partition is feasible by excluding all the letters after the first occurrence of $s_2^o[f(0) + 1]$. We then can have $l_0 r_0^1 \in (s_2^o[1..f(0)])^\diamond$. Now we apply Lemma 11 and get

$$l_0 r_0^1 r_0^2 \in (s_2^o[1..f(0)][f(0) + 1..g(0)])^\diamond = (s_2^o[1..h_1(0)])^\diamond$$

Then we prove the inductive case. Suppose the proposition holds for $i = k < n$, then we prove this holds for $i = k + 1$.

For the first part, we have known that $l_0 r_0 \cdots l_k r_k \in (s_2^o[1..h_1(k)])^\diamond$, then we also divide it into two cases.

- First, if $h_1(k) \geq f(k + 1)$, we know $l_0 \cdots l_{k+1} \in (s_2^o[1..f(k + 1)])^\diamond$ and $s_2^o[1..f(k + 1)] \in \text{pref}(s_2^o[1..h_1(k)])$. With the inductive hypothesis

$$l_0 r_0 \cdots l_i r_i \in (s_2^o[1..h_1(k)])^\diamond$$

, we know that all the letters in l_{k+1} have appeared in $l_0 r_0 \cdots l_i r_i$ and hence

$$l_0 r_0 \cdots l_i r_i l_{k+1} \in (s_2^o[1..h_1(k)])^\diamond = (s_2^o[1..h_2(k + 1)])^\diamond$$

- Second, if $h_1(k) < f(k + 1)$, we do the similar process as we prove $l_0 r_0 \in (s_2^o[1..h_1(0)])^\diamond$. We divide l_{k+1} into two parts $l_{k+1} = l_{k+1}^1 l_{k+1}^2$ such that l_{k+1}^1 contains only the letters from $s_2^o[1..h_1(k)]$ and $l_{k+1}^2 \in (s_2^o[h_1(k) + 1..f(k + 1)])^\diamond$, this is also feasible by excluding all the letters after the first occurrence of $s_2^o[h_1(k) + 1]$. With the division, we have

$$l_0 r_0 \cdots l_k r_k l_{k+1}^1 \in (s_2^o[1..h_1(k)])^\diamond$$

, hence we apply Lemma 11 and get

$$l_0 r_0 \cdots l_k r_k l_{k+1} \in (s_2^o[1..h_1(k)] s_2^o[h_1(k) + 1..f(k + 1)])^\diamond$$

For the second part, we have the similar process with an extra observation that $h_1(k+1) \geq h_2(k+1) \geq f(k+1)$. Then we divide this into two cases $h_1(k+1) \geq g(k+1)$ and $h_2(k+1) < g(k+1)$.

- In the first case we have $h_2(k+1) = h_1(k+1)$ from the inequality above, therefore r_{k+1} contains only the letters in $s_2^o[1..h_2(k+1)]$, hence

$$l_0 r_0 \cdots l_{k+1} r_{k+1} \in (s_2^o[1..h_2(k+1)])^\diamond = (s_2^o[1..h_1(k+1)])^\diamond$$

- In the second case, we also partition r_{k+1} into $r_{k+1} = r_{k+1}^1 r_{k+1}^2$. In the similar way, we have

$$l_0 r_0 \cdots l_{k+1} r_{k+1}^1 \in (s_2^o[1..h_2(k+1)])^\diamond \text{ and } r_{k+1}^2 \in (s_2^o[h_2(k+1)+1..g(k+1)])^\diamond.$$

By applying Lemma 11, we get

$$l_0 r_0 \cdots l_{k+1} r_{k+1} \in (s_2^o[1..h_2(k+1)] s_2^o[h_2(k+1)+1..g(k+1)])^\diamond = (s_2^o[1..h_1(k+1)])^\diamond$$

Finally, from equation (4.2), we have $l_0 r_0 \cdots l_n r_n \in (s_2^o[1..h_1(n)])^\diamond$. By the definition of $h_1(x)$ and our assumption that $s_1^o \in \text{pref}(s_2^o)$, we know that $h_1(n) = |s_2^o|$ and hence

$$l_0 r_0 \cdots l_n r_n \in (s_2^o)^\diamond \subseteq L^\diamond$$

□

4.2.2 Permutation-extended Pumping Closure

Definition 24. Given a word $s \in \Sigma^*$, a word $l \in \Sigma^*$ is called a *permutation* of s if and only if $|s| = |l|$ and there is a bijection $f : [1, |s|] \rightarrow [1, |s|]$ such that $\forall i \in [1, |s|] \cdot l[i] = s[f(i)]$.

We give an example to further illustrate this concept of permutation.

Example 8. Consider a word $s = abc$, we list all the permutations of s and the corresponding mappings below:

- $l_1 = abc$ where $f_1 = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3\}$.
- $l_2 = acb$ where $f_2 = \{1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 2\}$.
- $l_3 = bac$ where $f_3 = \{1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 3\}$.
- $l_4 = bca$ where $f_4 = \{1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 1\}$.
- $l_5 = cab$ where $f_5 = \{1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 2\}$.
- $l_6 = cba$ where $f_6 = \{1 \mapsto 3, 2 \mapsto 2, 3 \mapsto 1\}$.

For convenience, we define $\text{perm}(s) = \{l \mid l \text{ is a permutation of } s\}$ to collect all possible permutations of s . Since the number of the bijection f in the definition is finite, the cardinality of $\text{perm}(s)$ is also finite. This observation also means that it's possible to collect the permutations for all the prefixes for a given word.

Definition 25. Given a word $s \in \Sigma^*$, we construct a language call the *permutation collection* of s by

$$\text{perm}^{all}(s) = \sum_{l \in \text{pref}(s)} \text{perm}(l)$$

We still use the same example above to illustrate this concept.

Example 9. Consider a word $s = abc$, we list all the permutations of the prefix of s that is not euqal to s , and the corresponding mappings below:

- For prefix $ab \in \text{pref}(s)$, we have
 - $l_7 = ab$ where $f_7 = \{1 \mapsto 1, 2 \mapsto 2\}$.
 - $l_8 = ba$ where $f_8 = \{1 \mapsto 2, 2 \mapsto 1\}$.
- For prefix $a \in \text{pref}(s)$, we have
 - $l_9 = a$ where $f_9 = \{1 \mapsto 1\}$.
- For prefix $\varepsilon \in \text{pref}(s)$, we have
 - $l_{10} = \varepsilon$ where $f_{10} = \emptyset$.

The permutation of all the prefixes is therefore

$$\text{perm}^{all}(s) = \sum_{i=1}^{10} l_i$$

Definition 26. Given a word $s \in \Sigma^*$, we say a pumping closure $L^\diamond$ is a *permutation-extended* pumping closure of s if and only if L can be described as $L = L_1 + L_2$ where L_1 satisfies $L_1 = \text{perm}^{all}(s)$ and L_2 satisfies $\exists L' \subseteq \text{pref}(s), t \in \Sigma^* . L_2 = (\text{perm}(s) \cdot \{t\})$. Formally, we define a predicate

$$\begin{aligned} PE(L, s, t) = & \exists L_1, L_2 \subseteq \Sigma^* . L = L_1 + L_2 \wedge \\ & L_1 = \text{perm}^{all}(s) \wedge \\ & L_2 = (\text{perm}(s) \cdot \{t\}) \end{aligned}$$

Continuing with the example above, we give a concrete illustration of this concept.

Example 10. Consider a word $s = abc$ and a letter d , the following language L satisfies $PE(L, s, d)$:

$$L = \sum_{i=1}^{10} l_i + \sum_{i=1}^6 (l_i \cdot d)$$

Still, we put the main theorem first and then introduce some lemmas and prove the theorem in the end.

Theorem 14. *Given a word s , the interleaving shuffle is closed under any permutation-extended pumping closure of s , i.e.*

$$\forall s \in \Sigma^*, L \subseteq \Sigma^*, \exists t \in \Sigma^*. PE(L, t) \implies (\forall s_1, s_2 \in L^\diamond. s_1 \sqcup s_2 \subseteq L^\diamond)$$

Intuitively, to prove the closing property of permutation-extended pumping closure, we divide the possible composition of the pair (s_1, s_2) into five cases.

1. If both $s_1, s_2 \in (L_1)^\diamond$, and they come from a pumping closure of the permutation of the same prefix of s , i.e. $\exists l \in \text{pref}(s). s_1, s_2 \in (\text{perm}(l))^\diamond$.
2. If both $s_1, s_2 \in (L_1)^\diamond$, but they come from pumping closures of two different prefixes of s , i.e. $\exists l_1, l_2 \in \text{pref}(s). l_1 \neq l_2 \wedge s_1 \in (\text{perm}(l_1))^\diamond \wedge s_2 \in (\text{perm}(l_2))^\diamond$.
3. If both $s_1, s_2 \in (L_2)^\diamond$, i.e. $s_1, s_2 \in (\text{perm}(s) \cdot \{t\})^\diamond$.
4. If s_1 and s_2 come from two different sublanguages, without loss of generality, we express this as $s_1 \in L_1 \wedge s_2 \in L_2$.

We prove the first case by the following lemma.

Lemma 15. *Given a word $s \in \Sigma^*$ and two words $s_1, s_2 \in (\text{perm}(s))^\diamond$, we have $s_1 \sqcup s_2 \subseteq (\text{perm}(s))^\diamond$.*

Proof. We prove by construction. For a given shuffle $s' = l_0 r_0 \cdots l_n r_n \in s_1 \sqcup s_2$, we construct a permutation $u \in \text{perm}(s)$ such that $s' \in u^\diamond \subseteq (\text{perm}(s))^\diamond$. We now define a multiset $M = \{(s[1], \dots, s[|s|])\}$. And we have the following process to construct u .

1. Initialize u to be empty. Set counter $i = 1$ for the index of s' .
2. If $s'[i] \in M$, then append u with $s'[i]$ and remove one occurrence of $s'[i]$ from M .
3. If $u \notin \text{perm}(s)$, then increment $i = i + 1$ and repeat step 2.

The key idea of this procedure is that since $s_1, s_2 \in (\text{perm}(s))^\diamond$, they both contain all the occurrences of letters in s , which is M . Hence so does their interleaving shuffle s' . Then we traverse s' to collect all the occurrence of those letters and form a new word u , which is a permutation of s . If a letter is skipped at step 2, it means this letter has occurred before and hence can be generated by pumping operation. For all the remaining letters in s' after the loop, they also has occurred before in M because otherwise you cannot get them from pumping operation. This means these letters can also be generated by pumping operation. Till now, we have reform a u such that $s' \in u^\diamond$. Since $u \in \text{perm}(s)$, we also have $s' \in u^\diamond \subseteq (\text{perm}(s))^\diamond$. $\square$

Now let's look at the second case where s_1 and s_2 come from different prefixes.

Lemma 16. *Given a word $s \in \Sigma^*$, if there exists two different prefixes of s , namely $l_1, l_2 \in \text{pref}(s)$ and $l_1 \neq l_2$, then for two strings $s_1 \in (\text{perm}(l_1))^\diamond$ and $s_2 \in (\text{perm}(l_2))^\diamond$, we have $s_1 \sqcup s_2 \subseteq (\text{perm}^{\text{all}}(s))^\diamond$.*

Proof. We prove this in the same manner as in the previous lemma. Without loss of generality, we assume $|l_1| < |l_2|$, which means $l_1 \in \text{pref}(l_2)$. We now set multiset $M = \{l_2[1], \dots, l_2[|l_2|]\}$. Then for any $s' \in s_1 \sqcup s_2$, we still construct a permutation of l_2 using the same procedure in Lemma 15. Notice

$$M_1 = \{l_1[1], \dots, l_1[|l_1|]\} \subset M = \{l_2[1], \dots, l_2[|l_2|]\} \subset M' = \{s'[1], \dots, s'[|s'|]\}$$

This ensures that we still collect enough letters to form a permutation of l_2 , denoted as l'_2 . Hence we have

$$s' \in (l'_2)^\diamond \subseteq (\text{perm}(l_2))^\diamond \subseteq \text{perm}^{\text{all}}(s)$$

□

The third case are similar to the first case but with one shared suffix t . We address this common suffix by the following lemma.

Lemma 17. *Given two words $s, t \in \Sigma^*$, $(\text{perm}(s) \cdot \{t\})^\diamond$ is closed under interleaving shuffle.*

Proof. We prove this by dividing the composition of t into two cases.

The first case is that t is composed of only letters from s , i.e.

$$\forall i \in [1, |t|] . \exists j \in [1, |s|] . t[i] = s[j]$$

This case is trivial because we have $(\text{perm}(s) \cdot \{t\})^\diamond \subseteq (\text{perm}(s))^\diamond$ since appending t can be accomplished by applying a few pumping operations, i.e. $\text{perm}(s) \cdot \{t\} \subseteq \text{Pump}^{|t|}(\text{perm}(s))$.

The second case is that there exists at least one letter in t that does not appear in s , i.e.

$$\exists i \in [1, |t|] . \forall j \in [1, |s|] . t[i] \neq s[j]$$

Without loss of generality, we just assume this $i = 1$. Now for any $s_1, s_2 \in (\text{perm}(s) \cdot \{t\})^\diamond$ and $s' \in s_1 \sqcup s_2$, we divide s' into two parts $s' = s'_1 s'_2$. The partition requires $s'_2[1] = t[1]$ and s'_1 does not contain letter $t[1]$.

- For s'_1 , since the next letter is $t[1]$, s'_1 should contain all the letters of s . Since it is also an interleaving shuffle of s_1 and s_2 , this means $s'_1 = l_0 r_0 \dots l_i r_j$. Moreover, note that either $l_0 \dots l_i \in (\text{perm}(s))^\diamond$ or $r_0 \dots r_j \in (\text{perm}(s))^\diamond$, otherwise we aren't able to have $t[1]$ followed after according to the occurrence order words in $\text{perm}(s) \cdot \{t\}$. Without loss of generality, we assume $l_0 \dots l_i \in (\text{perm}(s))^\diamond$. Also note that $r_0 \dots r_j \in (\text{perm}(l))^\diamond$ where $l \in \text{pref}(s)$. This is exactly the second case we just proved in Lemma 16. Therefore we know $s'_1 \in (\text{perm}(s))^\diamond$.

- For $s'_2[1]$, we have $s'_2[1] = t[1]$, then we apply Lemma 11 and get $s'_1s'_2[1..1] \in (perm(s) \cdot \{t[1]\})^\diamond$. We then repeat this process: If the next letter in s'_2 is equal to the next letter in t , we append this letter to the base language of the pumping closure by applying Lemma 11. Otherwise we skip to the next letter of s'_2 . Eventually, we have $s'_1s'_2[1..k] \in (perm(s) \cdot \{t\})^\diamond$. Note the rest letters of s' must have appeared in $s'_1s'_2[1..k]$, therefore we know $s' \in (s'_1s'_2[1..k])^\diamond \subseteq (perm(s) \cdot \{t\})^\diamond$.

□

The last case can be proven in the same manner as the previous lemma. We can still partition the interleaving shuffle in the same way and append the letters of t in the same way. So we omit the proof here and only give the following lemma.

Lemma 18. *Given two words $s, t \in \Sigma^*$, for any two words $s_1 \in (perm(s) \cdot \{t\})^\diamond$ and $s_2 \in (perm^{all}(s))^\diamond$, $s_1 \sqcup s_2 \subseteq (perm(s) \cdot \{t\})^\diamond$.*

Finally, we prove the main theorem easily with the four lemmas above. We just need to break the distribution of s_1 and s_2 into four cases mentioned before and apply four lemmas accordingly. For convenience, we also omit the trivial proof of the theorem.

Chapter 5

Accumulative Tpestate System

In this chapter, we show the connection between pumping closure and tpestate systems. We prove that closing under the pumping operation is necessary for soundness without alias information and discuss the two special pumping closures to be sound without alias information.

In this chapter, we start with two motivating examples in Section 5.1. Then, we introduce the accumulative tpestate system in Section 5.2 and the two special groups of accumulative tpestates in Section 5.3. Finally, we give a brief discussion in Section 5.4 on the assumptions we made in this system, which could be left for future work.

5.1 Motivating Example

In this section, we informally introduce two examples of tpestate systems that make use of pumping closure to enforce some protocols for the use of methods.

Example 11. Consider a builder for some complex class *Car*. The simplified code is shown in Fig. 5.1. *Car* is a class that requires two external objects of the classes *Engine* and *ControlSystem*. Note that in the build method, there is some check that both the engine and control system should point to some non-null objects. Since you always set the component to a non-null object as long as you call the corresponding set method, this check actually checks if both set methods are called before the build method is called. Once the set methods are called, the construction will not fail. Therefore, it is fine to call them multiple times.

Example 12. Consider a snippet of code in Fig. 5.2 that passes an abstract syntax tree (AST) to the back-end of a compiler for code generation. However, the back-end requires the received AST to be instrumented with context information and type information. Therefore, we need to do name resolution and type checking before using emit() to pass the AST further down. Also, the AST cannot be type-checked without scope information. So, the name resolution must be done before the type checking is called. Since name resolution and type checking are idempotent, the number of calls to those methods does not matter.

```

public class Car {
    // Components of Car
    private Engine engine;
    private ControlSystem controlSystem;

    // Private constructor that takes a CarBuilder instance
    private Car(CarBuilder builder) {
        this.engine = builder.engine;
        this.controlSystem = builder.controlSystem;
    }

    ...

    // Static inner Builder class
    public static class CarBuilder {
        // Required components for Car
        private Engine engine;
        private ControlSystem controlSystem;

        ...

        // Setter for Engine
        public CarBuilder setEngine() {
            // Set engine if it has not been set, otherwise do nothing
            this.engine = engine == null ? new Engine("Default") : engine;
            return this;
        }

        // Setter for ControlSystem
        public CarBuilder setControlSystem() {
            // Set control system if it has not been set, otherwise no
            // nothing
            this.controlSystem = controlSystem == null ? new
                ControlSystem("Default") : controlSystem;
            return this;
        }

        // Build method to create an instance of Car
        public Car build() {
            // check if the parameters of car are all set, otherwise throw
            // exception
            if (engine == null || controlSystem == null) {
                throw new IllegalStateException("Engine and ControlSystem
                    must be set before building the Car");
            }
            return new Car(this);
        }
    }
}

```

Figure 5.1: Builder for Car.

```

abstract class AST {

    ...

    // Name resolution attach each declared variable with it's declared type
    public void name_resolution(){
        ...
    }

    // Type checking checks if all variable uses respect their types
    // Run into error if name resolution is not done
    public boolean type_checking(){
        ...
    }

    // Omit the type checked AST to backend
    // Produce faulty code if AST is not type checked
    public void emit(){
        ...
    }

    public static void compile(){

        ...

        AST.name_resolution();

        // resolution must be called before type checking.

        AST.type_checking();

        // before omit is called, resolution and type checking should be
        // called in order.

        AST.emit();
    }
}

```

Figure 5.2: Compiler Front-end.

We extract two requirements from these examples. First, we need some static analysis to (soundly) verify if this protocol is enforced. This leads us to a flow-sensitive analysis to keep track of the sequence of methods that could have been called on the variables, which is matched well to the concept of a tpestate system. Second, we note that there is some good property for the two protocols above, which is that you don't lose access to the method you have called. In other words, you only gain access to call the method during the execution. For the builder, you can call the set methods arbitrarily after you have called it once; none of those calls violate the protocol. The same appears for the compiler case. You can always re-do the name resolution after your first call to it. The only small difference is that it also requires the order of the method calls to gain access. You must call the name resolution and type checking in the right order.

5.2 Accumulative Tpestate System

In this section, we define a special kind of tpestate system that focuses on the protocols that can be expressed by the pumping closure, which is inspired by previous work [33]. Then we discuss the need for alias information for the soundness of such kinds of tpestate systems. Intuitively, we show that the pumping operation is necessary to eliminate the need for alias information, i.e. $TS_{\emptyset, \pi}$ is sound only if the language s from π is closed under the pumping operation. Then we discuss two extra properties that, both of them combined with the pumping closure, are sufficient to be sound without alias information. We also showcase some pumping closures that are not sound without alias information. We give a more detailed comparison between our definition and the one in previous work in Section 6.1.2.

5.2.1 Definition

We introduce the definition of the accumulative tpestate system. We start with a previous example 11. Note that once some set method is called over the *CarBuilder*, you can call it arbitrarily. Therefore, it is possible to relax the protocol to allow arbitrary calls after the first call of some method. For example, the legal method calls on a *CarBuilder* object *builder* can be the following (we use method chaining for readability).

$$builder.setEngine().setCotrolSystem().build() ,$$

or

$$builder.setEngine().setCotrolSystem().setEngine().build() .$$

Let us now call this observation *accumulative property* without formal definition. This accumulative property can also be easily described in regex. Let $\llbracket setEngine \rrbracket = a$, $\llbracket setControlSystem \rrbracket = b$, and $\llbracket build \rrbracket = c$. For the class, we also need a , b and c to be called at least once and c to be called after a and b have been

called, which can be expressed as $s = abc + bac$. Now, we need to add more elements to s according to the accumulative property. The protocol of class *CarBuilder* can then be expressed as $s' = aa^*b(a+b)^*c(a+b+c)^* + bb^*a(a+b)^*c(a+b+c)^*$.

If we look at this extension from s to s' , we would notice that this is exactly what we have defined as pumping closure. This leads to the following definition of the accumulative tpestate system.

Definition 27. A tpestate system $TS_{O,\pi}$ is called *accumulative tpestate system* if and only if the language s from π can be expressed as a pumping closure.

An immediate corollary of this definition is a necessary condition for the accumulative property.

Corollary 19. A tpestate system $TS_{O,\pi}$ is accumulative only if the protocol of C is closed under the pumping operation.

This is an efficient way to determine whether a tpestate system is not accumulative. Given a protocol $\pi = (\llbracket \cdot \rrbracket, r)$, as long as there exists a word $w \in r$ such that there exists a pumping word $w' \in Pump(w)$ that is not accepted by the protocol, i.e. $w' \notin r$, we could say that any tpestate system of the protocol π is not accumulative.

5.2.2 Need of Alias Information

With the formal definition of accumulative tpestate system, we now introduce the first property of accumulative tpestate system. The property is related to the need for alias information in accumulative tpestate system. We have included the use of *MayOracle* in the typing rules. First of all, the soundness of the oracle can imply the soundness of the tpestate system.

Theorem 20. A tpestate system $TS_{O,\pi}$ is sound if the oracle O is sound.

Proof. This is proven by co-induction on the operational semantics (Fig. 3.8) and typing rules (Fig. 3.9). $\square$

But what about the opposite: Could the tpestate system be sound even without the oracle being sound? The answer is yes, and we don't even need the oracle for the tpestate system to be sound. To fully answer this question, we first define a tpestate system that has no alias information.

Definition 28. A tpestate system $TS_{O,\pi}$ is called *without alias information* if the oracle always returns nothing, i.e. $O = \emptyset$.

It is rather trivial to show that a tpestate system without alias information may not be sound in general. The file example (Fig. 3.5) at the beginning of this section would not be sound without alias information.

It's natural that we want to restrict this group of tpestate systems without alias towards the sound ones. One common approach is to find the necessary or sufficient conditions for the soundness of tpestate systems without alias information. In the following, we prove that the accumulative tpestate system is the only kind of tpestate system that can be soundly checked without alias information, i.e. pumping-closedness is a necessary condition.

Theorem 21. *If a tpestate system without alias information, $TS_{\emptyset, \pi}$, is sound, then it is an accumulative tpestate system.*

Proof. We prove this by contradiction. Suppose that there is a tpestate system of protocol π which is not accumulative but can be checked without alias information. From the corollary 19 and the Definition 4.1, we know that there must exist a word $s \in r$ and one of its pumping words $w \in Pump(s)$, such that $w \notin r$. Without loss of generality, let $s = l_1 \cdots l_n$ and $w = l_1 \cdots l_n l_1$, and let the methods defined in C have the same order as the letters, i.e. $\llbracket m_1 \rrbracket = l_1 \cdots \llbracket m_n \rrbracket = l_n$. Now we construct a program statement S of class C as follows:

```
x_1 = x_2;
x_1.m_1();
...
x_1.m_n();
x_2.m_1();
...
x_2.m_n();
return
```

By applying the typing rules, the program type checks under tpestate system $TS_{\emptyset, \pi}$. The tpestates of x_1 and x_2 are both $D_s(r)$. By the assumption that $s \in r$, we get $\varepsilon \in D_s(r)$.

However, if we execute the program under our operational semantics, we get stuck in the middle because there is no semantics to apply. Let $\sigma(x_1) = \sigma(x_2) = \&l$ after $x_1 = x_2$. Then we have the same $\tau(l) = D_s(r)$ for the object $o = h(l)$ after $x_1.m_n()$. By the property of the derivative, we know that $D_w(r) = \emptyset$. During the reduction of $x_2.m_1()$, it computes $D_{l_1}(D_s(r)) = D_{sl_1}(r) = D_w(r) = \emptyset$, which fails the empty check and blocks the execution of the program. To this point, we have a program with statement S that type checks but has undefined behavior w.r.t. the operational semantics. This result contradicts the assumption at the beginning that this tpestate system can be checked soundly. Hence, the theorem is proved. $\square$

In summary, the theorem above shows that the accumulative property is a necessary condition to remove the need for alias information. This greatly reduces the scope when considering the soundness without alias information. The next question arises is the other direction of the problem: can every accumulative tpestate system be checked soundly without alias information? Unfortunately, the answer is not always. In the next section, we divide the results into smaller groups by adding more constraints. Then we show two kinds of accumulative tpestate systems that can be proved to be checked soundly without alias information, and an example of an accumulative tpestate system that cannot be checked without alias information.

5.3 Special Accumulative Typestate System

We now look at an example that shows that an accumulative typestate system cannot always be soundly checked without alias information.

Example 13. Consider an accumulative typestate system for class C with a protocol π with language $r = (abc + cab)^\diamond \subseteq \Sigma^* = \{a, b, c\}^*$ and a method list $\bar{M} = a, b, c$. The corresponding map is $\llbracket a \rrbracket = a, \llbracket b \rrbracket = b, \llbracket c \rrbracket = c$. Consider this following program

```
x_1 = x_2;
x_1.a();
x_2.c();
x_2.a();
x_2.b();
x_1.b();
x_1.c();
```

We can compute the pumping closure r to be

$$r = aa^*b(a+b)^*c(a+b+c)^* + cc^*a(a+c)^*b(a+b+c)^*$$

The sequence of method call on the object that x_1 and x_2 points to is $o : \textcolor{red}{a}\textcolor{blue}{c}\textcolor{blue}{a}b\textcolor{red}{b}c$. Note that $\textcolor{red}{a}ccabbc \notin r$ because c cannot be called immediately after a without calling b .

However, for each variable x_1 and x_2 , we extract their sequence of method calls as $x_1 : abc$ and $x_2 : cab$. Note that both are accepted by r , therefore this program type checks without alias information. This conflict indicates that this pumping closure cannot be check soundly without alias information.

There are two observations from this example. The first one is that this accumulative typestate system cannot be checked soundly without alias information. The other one is that any combination of sequences of method calls from two aliases is an interleaving shuffle of those two sequences. The first observation serves as evidence that accumulation cannot imply soundness without alias information. The second observation links the soundness without alias information to the closure under interleaving shuffle. If the problem of proving the soundness of typestate systems without alias information can be reduced to the problem of proving the closing property of the corresponding protocol under interleaving shuffle, we would immediately get two kinds of typestate systems that are sound without alias information.

We have observed one such case where the combination of the sequences of two aliases is the interleaving shuffle of those two sequences. Now we just need to prove that this is always the case.

Lemma 22. *Given a typestate system over class C with protocol π of language $r = L^\diamond$ and method list $\bar{M}$, for any pair of aliases e_1 and e_2 referring to object o , and respectively the call sequence s_1 and s_2 , the call sequence of o is an interleaving shuffle of s_1 and s_2 .*

The proof is straightforward. First, we partition the method calls of e_1 and e_2 into smaller groups, each containing only the method calls to one variable, and the adjacent groups belong to different variables. Then, we map each group to r_a and l_b in Definition 21 and hence show that this fits into the definition.

Now, we have reduced this problem of soundness without alias information to a shuffle-closed property. Since we have proven two special kinds of pumping closures that are closed under interleaving shuffle, we expand them into two kinds of tpestate systems.

Definition 29. An accumulative tpestate system is called *prefix-included* tpestate system, denoted as $PI_{s,\pi} = TS_{\emptyset,(\llbracket \cdot \rrbracket, L_s^\diamond)}$ if and only if L_s satisfies $\forall s' \in L_s. s' \in \text{pref}(s)$.

Definition 30. An accumulative tpestate system is called *permutation-extension* tpestate system, denoted as $PE_{s,t} = TS_{\emptyset,(\llbracket \cdot \rrbracket, L(s,t)^\diamond)}$ if and only if $L^\diamond$ is a permutation-extension pumping closure of s , i.e. $L(s,t) = \text{perm}^{all}(s) + (\text{perm}(s)\{t\})$.

Combined with lemma 22 linking aliases and interleaving shuffle, we show these two special kinds of accumulative tpestate systems can be soundly checked without alias information.

Theorem 23. *Prefix-included accumulative tpestate system $PI_{s,\pi}$ is sound without alias information.*

Theorem 24. *Permutation-extension accumulative tpestate system $PE_{s,t}$ is sound without alias information.*

The proofs of these two theorems are similar. We put the proof of the first theorem for prefix-included accumulative tpestate system here. The proof for the other theorem Thm. 24 only requires changing the assumption from *prefix-included* to *permutation*.

Proof. Since two expressions do not interfere with each other in terms of protocol conformance if they refer to different objects, we only consider the expressions pointing to one single object. Suppose we have a Prefix-included accumulative tpestate with class C and protocol $(\llbracket \cdot \rrbracket, L_s^\diamond)$. Let $e_1, \dots, e_n$ be the aliases of an object o of C and let the corresponding call sequences be $s_1, \dots, s_n$. To prove the soundness of our target type system, we also assume that $s_1, \dots, s_n \in L_s^\diamond$. We prove the theorem by induction on the number n .

For the base case $n = 1$, the result is trivial because there is no alias, and from the soundness assumption, we know that o follows the protocol.

For the inductive case $n = k + 1$, we have two inductive hypotheses. The first hypothesis states that the call sequences of $e_1, \dots, e_{k+1} \in L_s^\diamond$. The second inductive hypothesis states that the tpestate system is sound for $e_1, \dots, e_k$, meaning the status of conformance of o is in $L_s^\diamond$ ignoring e_{k+1} .

Based on the second hypothesis, we rename all expressions $e_1, \dots, e_k$ to one name E and denote the corresponding call sequence as $s_E \in L_s^\diamond$. The new program

only contains two aliases of o , and this new program is equivalent to the previous one. Then, we apply the lemma 22 to the pair of aliases E and e_{k+1} . Then we know the call sequence of o is an interleaving shuffle of s_E and s_{k+1} , denoted as s' . We also know from the inductive hypotheses that $s_E, s_{k+1} \in L_s^\diamond$. Since the protocol is prefix-included, we know that the interleaving shuffle of its elements is still in the language, i.e. $s' \in r^\diamond$. This means that o also follows the protocol after considering e_{k+1} . $\square$

5.4 Discussion

Note that we have two quite strong assumptions about the operation in this typestate system and the underlying FSM protocols. We assume that the only operation considered as a transition in our typestate system is a method call, and the protocol has correctly encoded the whole usage of the method into an FSM protocol.

Since we have the correct and desired use of the method and APIs, we focus on the method calls in this thesis. However, this is not the limit of the typestate system's capability. More instructions can be taken into account in the typestate system to improve the expressiveness and granularity of the typestate system. Meanwhile, the protocol conformance check still remains the same, as the protocol is still a regular language. This leads to some possible future work to encode linear and uniqueness types by including the assignment statements as operations.

The second assumption is actually a direct consequence of the first one. The method body is intentionally ignored because we cannot make any transitions with any other kind of statements like variable assignments. Therefore, we cannot really gain any information by going into the method body at a certain call site. This leads to some other future direction, like having a more fine-grained analysis that takes the method body into account, which may extend the expressiveness of the protocols. For instance, we can have a non-deterministic transition based on the context at the call site.

Chapter 6

Related Work

In this chapter, we discuss related works in soundness without alias information, typestate system and behavioral types that check protocols.

6.1 Other Attempts to remove alias information

Typestate systems traditionally track object states and enforce state transitions while ensuring soundness in the presence of aliasing. However, several approaches have been developed that achieve soundness without explicitly tracking alias information. In general, these approaches can be divided into two groups. One group restricts the protocols and proves that soundness is guaranteed for protocols from this class, even without alias information (Sections 6.1.1 and 6.1.2). Another group restricts the use of data (variables, expressions, and/or references) to eliminate the possibility of aliasing (Sections 6.1.3, 6.1.4, and 6.1.5).

6.1.1 Heap-Monotonic Typestate

Based on previous work [16], Fähndrich and Leino introduced *heap monotonic typestates* [17], a restricted class of typestate systems that remain sound in the presence of aliasing. The key insight is to require that an object’s state can only evolve *monotonically*—once an object attains a typestate, it never regresses to a weaker typestate. In other words, all state changes strengthen the object’s invariants over time, never invalidating previous state assumptions. This monotonicity guarantees that if a reference is known to have a certain typestate, then *no future heap updates can falsify that knowledge*. Consequently, aliasing is no longer dangerous: even if two aliases refer to the same object, any state-changing operation via one alias can only advance the object’s typestate, so an alias will never observe an weaker state. This yields soundness without requiring an alias analysis. Jagadeesan *et al.* [28] later provided a formal type system for monotonic typestates in an imperative object calculus, proving type soundness (progress and preservation) for a language with object state evolution under relaxed aliasing assumptions.

The similar monotonicity can also be built upon accumulation type system. The DFA is a better way to illustrate it. The pumping closure has the structure that whenever you reach a non-error state s by transition t , then under all the reachable state from s , t will transit to a non-error state. In other words, the transition that leads to error state is monotonically decreasing. In summary, we could say that our accumulative tpestate system is an instance of heap monotonic tpestate.

6.1.2 Accumulation Analysis

A very recent advancement is the identification of *accumulation tpestates* and their associated analysis by Kellogg *et al.* [33]. An accumulation tpestate specification is one that can be enforced soundly by only tracking which operations have been performed on an object, without any alias information. The analysis works by conservatively *accumulating* the set of mutating operations (method calls, field writes, etc.) that have occurred on each object, and using this history to determine the object's current tpestate. If all operations required by the tpestate specification have been seen (accumulated) before a given usage, then that usage is safe. For example, if a specification requires that a File must be open before calling `read()`, an accumulation analysis would simply check that an `open()` call appears on that File's history before any `read()` call.

This work is tightly connected to this work. The *error-inducing sequence* and *accumulation tpestate system* in accumulation analysis are very similar to our *pumping closure* and *accumulative tpestate system*. We translate the two concepts into our setting of protocol conformance.

Definition 31. Let $\pi = (\llbracket \cdot \rrbracket, r)$ be a protocol of class C . A word $w = l_1 \cdots l_n \in \Sigma^*$ is called an *error-inducing sequence* of language r if and only if

$$D_{l_1 \cdots l_{n-1}}(r) \neq \emptyset \wedge D_w(r) = \emptyset$$

Intuitively, an error-inducing sequence is such a word that does not violate the protocol before the final letter l_n is appended. This definition is different from the pumping word in such a way that it is a property of the words that are not accepted by the language, i.e. a property of the complement of r .

Definition 32. A tpestate system TS of protocol $\pi = (\llbracket \cdot \rrbracket, r)$ is called an *accumulation tpestate system* if and only if the error-inducing sequence of r is closed under the (contiguous and non-contiguous) subsequence operation while fixing the last letter of the sequence, i.e.

$$\forall s \in \Sigma^*, l \in \Sigma, s' \in \text{Sub}(s) \cdot D_s(r) \neq \emptyset \wedge D_{sl}(r) = \emptyset \implies D_{s'}(r) \neq \emptyset \wedge D_{s'l}(r) = \emptyset$$

where $\text{Sub}(s) = \{w \mid w \text{ is a subsequence of } s\}$.

Intuitively, the accumulation tpestate is similar to accumulative tpestate in the way that the former is saying that removing letters from a rejected word will

remain rejected, while the latter is saying that adding letters to an accepted word will remain accepted. This intuition may look like they are equivalent. However, actually there are subtle differences between them. The key reason is the extra constraint of error-inducing sequence that requires the whole word to be rejected, but removing the last letter will cause the word to be accepted. We give an example of an accumulative tpestate system that is not an accumulation tpestate system.

Example 14. Consider an accumulation tpestate system whose protocol language is

$$r = (abcd)^\diamond = aa^*b(a+b)^*c(a+b+c)^*d(a+b+c+d)^*$$

$w = abad$ is an error-inducing sequence of r because

$$D_{abad}(r) = \emptyset \wedge D_{aba} = (a+b)^*c(a+b+c)^*d(a+b+c+d)^* \neq \emptyset$$

Then consider word ba which is a subsequence of aba . We have

$$D_{bad}(r) = \emptyset \wedge D_{ba}(r) = \emptyset$$

The second part violates Def. 31.

Moreover, note that the language r above is a prefix-included pumping closure, which means that the accumulative tpestate system in the example is actually sound without alias information, which also breaks the claim in the previous work stating that accumulation tpestate systems are exactly the tpestate system that can be checked without alias information.

6.1.3 Linear and Uniqueness Typing

Linear and uniqueness type systems enforce alias avoidance at the language level, guaranteeing that at most one reference (or one mutable reference in the case of affine types) exists for a given object at any time. This originates from substructural type theory, where resources must be used in a linear fashion (consumed exactly once) or under uniqueness constraints. By preventing multiple aliases, these type systems make tpestate verification inherently simpler and sound, since no external alias analysis is needed to ensure that unseen aliases haven't invalidated an object's state. For example, if an object's type changes from *File_{closed}* to *File_{open}* upon calling an `open()` method, a linear type system ensures no other alias of that object exists to call `open()` again or use the file while it's closed – the single owning reference must respect the tpestate protocol.

In practice, linear/unique type disciplines have been highly effective. Rust's ownership model (affine types) is one of the most well-known examples: Rust's compiler tracks ownership and borrowing of values to ensure memory safety and protocol correctness (for instance, a file cannot be used after it is closed or deallocated, as the type system prevents any such use-after-close). This design has been proven sound in formal models (e.g., RustBelt [31]) and demonstrates that linear/affine types can enforce tpestate-like properties (e.g., ensuring a resource is

initialized before use and not used after disposal) without any runtime alias checks. Recent work also shows how linear types can be integrated with verification tools: Li *et al.* [35] combine linear types with SMT-based verification, treating unaliased objects as pure values to simplify reasoning. In their case, linear types drastically reduce the need to reason about aliasing, since the type system guarantees exclusive access, and they report improved verification performance and fewer proof obligations when 91% of a system’s objects were made linear.

Overall, linear and uniqueness type systems achieve sound typestate enforcement by design: an object’s state-changing operations are permitted only through a single reference, so no hidden alias can violate the expected state, obviating the need for external alias-tracking mechanisms.

6.1.4 Capabilities-Based Typestate Systems

Capabilities-based typestate systems build on the idea of controlling aliasing and permissible operations through access permissions or capabilities associated with object references. Instead of outright forbidding aliases, these systems assign permissions (unique, shared read-only, immutable, etc.) to each reference, and the typestate of an object is coupled with the permissions of the references that alias it. A capability can be seen as a token that grants the right to perform certain state transitions on an object. The uniqueness is a suitable target to encode by capability. For example, having an unique capability might be required to move an object from state S_1 to S_2 , whereas a read-only capability might allow reading the object’s data only if the object is in a state where reads are legal.

By making these capabilities part of the static type, the type checker ensures that aliasing cannot lead to illegal state transitions: if a reference has a capability to perform an operation, no other alias can simultaneously exist with a conflicting capability. The typestate transitions typically require a full permission, which means that no other active alias can hold that full permission at the same time. This discipline avoids the need for global alias analysis because the type system locally ensures that any potential alias either does not exist or does not undermine the typestate invariant. For example, consider a capability-based system where an object in state Closed can be opened only if the caller holds a unique Closed-state capability for that object. If there were any aliases, they could only exist with weaker permissions (like read-only), which would not allow them to perform an open operation. Thus, the state transition is sound.

In summary, capability-based typestate systems use type-encoded permissions to statically enforce alias control and state correctness. They are sound typestate systems because the typestate invariants are maintained by the capability rules (only certain capabilities can cause certain state transitions, and conflicting capabilities cannot coexist). By construction, this obviates the need for external alias tracking – any aliasing that does occur is explicitly accounted for by restricted permissions, and the type system prevents unsanctioned interactions between aliases and object state.

6.1.5 Separation Logic-Based Approaches

Separation logic-based approaches tackle typestate verification from the angle of program logic and formal verification. Separation logic [45, 41] provides a way to reason about memory such that aliasing is handled implicitly by the logic’s semantics: the logic’s central concept, the separating conjunction ($*$), specifies that two assertions hold for disjoint parts of memory. This built-in notion of disjointness means that if a piece of state (an object’s fields) is needed to satisfy a certain predicate (e.g., an object being in a particular state), the logic ensures no other part of the program can simultaneously be acting on that same piece of state unless explicitly allowed. In other words, separation logic allows one to enforce protocols by ownership transfer or sharing disciplines, rather than global alias analysis.

For typestates, one can define a state predicate for each state of an object (for example, $Closed(o)$ and $Open(o)$ could be predicates describing object o ’s layout or invariants in those states). To call a method that requires the object to be open, the precondition in the Hoare logic would require the caller to possess the $Open(o)$ predicate (and maybe exclusive ownership of o). If the object was closed, the caller must first execute an opener method whose postcondition transfers the predicate from $Closed(o)$ to $Open(o)$ (representing the state change). Crucially, if any alias to o existed elsewhere in the program, it could not simultaneously hold the $Closed(o)$ or $Open(o)$ predicate because separation logic would not allow two distinct threads or parts of the program to own the same predicate for o at once. Thus, the logic itself prevents unsynchronized aliases from violating the typestate: an alias would have to go through the same proof obligations to access or modify o ’s state.

Modern separation logics like TaDA [12] and Iris [32] even allow reasoning about concurrent stateful protocols (typestates in a concurrent setting) by combining separation logic with protocol state machines. TaDA, for example, introduces protocol combinators in the logic to specify atomic transitions on shared objects, and it ensures that threads must possess the appropriate token (capability) to perform a transition, very much like typestate transitions, but in the logic. Because only one thread at a time can hold the protocol’s token for a given object, this again mimics the effect of alias control (only the holder can act on the object’s state).

In summary, separation logic approaches achieve sound typestate verification by using ownership and permissions to manage aliasing: they do not require a separate alias analysis because the logic’s assertions inherently track who has the right to access an object’s state. The trade-off is that the developer or verifier must set up the correct predicates and invariants, but many tools and frameworks now assist with this, and they have shown success in real case studies like verifying correctness of Java iterators, file streams, and complex data structure protocols in projects such as VeriFast and Ironclad.

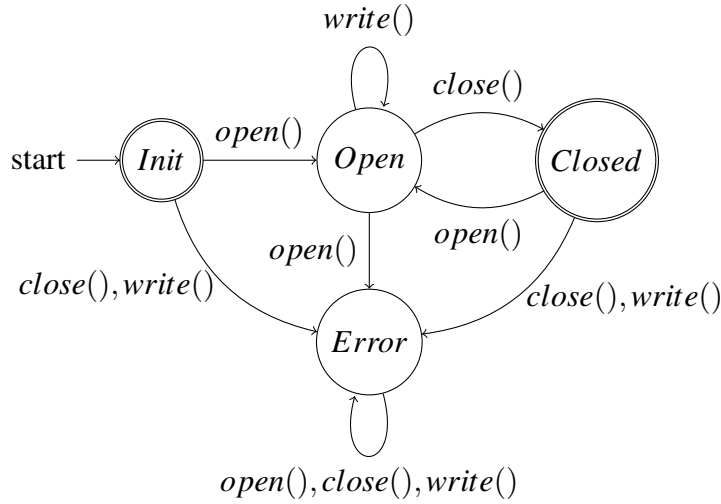


Figure 6.1: DFA of File Object

6.2 Typestate Systems in Other Form

Because typestates essentially describe allowable sequences of operations (state transitions) for objects, they can be formalized in various ways. We discuss several representations of typestate systems and how they relate to one another, including finite state machines, logical transition relations, session types, and effect systems. Each of these offers different theoretical strengths and weaknesses in specifying and verifying protocols.

Typestate systems can be formally described using various representations, each offering different advantages in terms of expressiveness and verification efficiency. We present each approaches in the following subsections.

6.2.1 Finite State Machines (FSMs) and Deterministic Finite Automata (DFAs)

The classical view of typestate is as a finite-state automaton (often deterministic) for each protocol [14, 30, 19, 33]. In this view, each object’s type is associated with a state machine where states represent abstract states of the object, and transitions are triggered by method calls, or more generally, operations. For example, a File object might have a typestate automaton with states *Closed*, *Open*, and an error state, and transitions for operations *open()*, *close()*, and *write()* that move the object between states, as shown in Fig. 6.1.

This DFA (deterministic finite automaton) model of typestates is intuitive and maps directly to state diagrams used in software design. Its theoretical strength lies in decades of automata theory: many verification problems (such as typestate safety—ensuring no invalid transition is taken) reduce to reachability or language inclusion checks on these automata, which are well-understood and, for finite-state properties, decidable. The simplicity of DFAs also makes certain analyses

straightforward and fast (e.g., dataflow analysis can track the current state of each object in the abstract state machine). The DFA model has exactly the same expressiveness as regular expressions even though they are equivalent. The state is specified explicitly, often with descriptive names that help the user to have an intuition about the protocol.

6.2.2 Transition Relations

An alternative representation of typestate is to embed state transition logic into the type system as logical predicates and relations [2, 21]. Instead of drawing an explicit automaton, one can specify for each operation (normally method calls) a transition relation that describes how that operation changes the object's state. For example, one might declare method `close()` as:

```
public void close() [Open » Closed] ...;
```

A series of such specifications implicitly defines the transition system of the typestate automaton. The strength of using transition relations is greater modularity and maintainability. Only one or a few transitions need to be specified in each method declaration indicating the use of that single method, instead of defining the whole structure at once. However, the downside is that minor changes to the transition system may heavily change the shape of the system, which may require more heavy-weight verification techniques as opposed to the whole description of DFA-based or regex-based typestate. In addition, global state storage and consistency checks between transitions increase the overhead of the analysis.

6.3 Other Behavioral Types

The term “behavioral type system” encompasses systems that try to verify that programs adhere to specified behaviors (communication patterns, sequencing of actions, etc.) as opposed to just data shape or simple type compatibility [3]. Compliance of some protocols is the major focus under this category; therefore, typestate is one of the prime examples of behavioral types. In the following, we introduce some other prime examples of behavioral types with a similar focus.

6.3.1 Session Types

Session types [25] are a type-based formalism originally developed for specifying communication protocols (especially in message-passing concurrent systems), but they are closely related to typestates and can be seen as an alternative view of protocol typing. A session type describes the sequence of messages (send or receive actions) that an endpoint in a communication can engage in, effectively forming a state machine for communication channels. For example, a simplified two-party protocol might be described by a session type:

`?login.!(password).?ok.end`

meaning “first receive a login, then send a password, then receive an ok, then terminate.” This is analogous to a typestate automaton for a channel. This can also be seen as a definition of transition relations. If a session type is well-formed, i.e. every $?S$ has a corresponding $!S$ and vice versa, then every pair of $(?S, !S)$ can be seen as a transition relation, therefore implicitly defines some state. Compared to object typestate systems, session types focus on communications. For well-formed session type specifications, one can often prove deadlock-freedom or that sessions will complete without participants indefinitely waiting on each other.

In many session types’ setting, a channel’s usage is linear: each session is owned by exactly two endpoints (in the binary case) and no aliasing of an endpoint is allowed, so that the communication protocol progresses in a coordinated fashion. This linearity avoids aliasing issues by design, which is similar in spirit to typestate approaches that require uniqueness or linearity. The development of *multiparty session types* (MPST) by Honda *et al.* [26] and Bettini *et al.* [5] generalizes binary (two-party) session types to protocols involving multiple parties and complex interaction patterns.

Researchers have drawn connections between session types and typestates. Simon *et al.* [22] showed that an object’s typestate protocol can be seen as a session between the object and its client code. Hannes *et al.* [47] gave a full formalization of session type based on typestate system.

6.3.2 Effect Types

Effect systems [51, 38] are an extension of type systems that, alongside the usual type of each expression, also compute an “effect” (some abstract description of the computation). In the context of typestate and protocols, an effect system can be employed to track protocol-related actions. For example, one could define an effect that represents “resource X has been opened” or “operation Y has been performed”. A function that opens a file might have an effect $Opened(File)$, and a function that requires a file to be open could be checked to ensure it has that effect in its calling context.

The strength of effect systems is the expressiveness of the side-effects; they augment the type of each function or method with additional information (the effect) that the compiler can check and propagate. Unlike a monolithic state machine, effects can be locally checked and combined following algebraic rules (typically a commutative monoid or similar). For instance, an effect system might say function f has effect E_f and function g has effect E_g ; if we compose $f;g$, the total effect is $E_f \cdot E_g$. The theoretical downside of effect systems is that if the effect algebra becomes too expressive (for example, trying to encode an arbitrary state machine as an effect), the system can lose the simplicity that makes it tractable. Most effect systems used in practice (like Java’s checked exceptions, or simpler read/write effects for memory) are therefore intentionally limited in expressiveness.

Recent research has shown a close correspondence between certain effect systems and session/typestate systems. Orchard *et al.* [42] demonstrated that session types are expressive enough to encode a conventional effect system for state, by giving a translation from a lambda calculus with a type-and-effect system into a π -calculus with session types. Militao *et al.* [40] showed that typestate system can be seen as an effect system where the effect of a method call is to change the typestate (encoded as linear resource consumption/production). While being equivalent, effect systems provide a useful alternate lens for typestate: instead of thinking in terms of states, one thinks in terms of actions (effects) and ensures the necessary actions have occurred by the time certain points in the program are reached.

Chapter 7

Conclusion

In this thesis, a novel formalization of the typestate system is introduced, utilizing regular expressions to encode the protocol and regular expression derivatives to verify protocol compliance. Furthermore, a subset of the typestate system, termed the accumulative typestate system, is identified by imposing restrictions on the protocol.

The first restriction is articulated as the closure of the regular language under a newly defined operation called pumping word. It is demonstrated that the accumulative typestate constitutes a superset of the typestate system that can be reliably checked without recourse to alias information. Two subsets of the accumulative typestate system are identified by two extra constraints. They are proved to be sound without alias information, as the protocols within these subsets are closed under the interleaving shuffle operation, which encodes all interactions between the call sequences of aliases.

References

- [1] Jonathan Aldrich, Robert Bocchino, Ronald Garcia, Mark Hahnenberg, Manuel Mohr, Karl Naden, Darpan Saini, Sven Stork, Joshua Sunshine, Éric Tanter, et al. Plaid: a permission-based programming language. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 183–184, 2011.
- [2] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1015–1022, 2009.
- [3] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J Gay, Nils Gesbert, Elena Giachino, Raymond Hu, et al. Behavioral types in programming languages. *Foundations and Trends® in Programming Languages*, 3(2-3):95–230, 2016.
- [4] Alen Arslanagić, Pavle Subotić, and Jorge A Pérez. Bit-vector typestate analysis. *Formal Aspects of Computing*, 35(3):1–36, 2023.
- [5] Lorenzo Bettini, Mario Coppo, Loris D’Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *International Conference on Concurrency Theory*, pages 418–433. Springer, 2008.
- [6] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. *ACM SIGPLAN Notices*, 42(10):301–320, 2007.
- [7] Kevin Bierhoff and Jonathan Aldrich. Plural: checking protocol compliance under aliasing. In *Companion of the 30th international conference on Software engineering*, pages 971–972, 2008.
- [8] Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A Naeem. Collaborative runtime verification with tracematches. In *Runtime Verification: 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers 7*, pages 22–37. Springer, 2007.
- [9] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964.

- [10] Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and assets for safer blockchain programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 42(3):1–82, 2020.
- [11] Silvia Crafa and Luca Padovani. The chemical approach to typestate-oriented programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(3):1–45, 2017.
- [12] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In *ECOOP 2014—Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings 28*, pages 207–231. Springer, 2014.
- [13] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004.
- [14] Matthew B Dwyer and Rahul Purandare. Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 124–133, 2007.
- [15] Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäfer, Aritra Sengupta, and Willem Visser. Rapid: checking api usage for the cloud in the cloud. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1416–1426, 2021.
- [16] Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 13–24, 2002.
- [17] Manuel Fähndrich and K Rustan M Leino. Heap monotonic typestates. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, pages 58–72, 2003.
- [18] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *International conference on formal verification of object-oriented software*, pages 10–30. Springer, 2010.
- [19] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–34, 2008.
- [20] Jeffrey S Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12, 2002.

- [21] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of tpestate-oriented programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(4):1–44, 2014.
- [22] Simon J Gay, Nils Gesbert, António Ravara, and Vasco T Vasconcelos. Modular session types for objects. *Logical Methods in Computer Science*, 11, 2015.
- [23] Radu Grigore, Dino Distefano, and Nikos Tzevelekos. Automatic compositional checking of multi-object tpestate properties of software. In *Principles of Verification: Cycling the Probabilistic Landscape: Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part I*, pages 3–40. Springer, 2024.
- [24] Christian Haack and Clément Hurlin. Resource usage protocols for iterators. *Journal of Object Technology*, 8(4):55–83, 2009.
- [25] Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming*, pages 122–138. Springer, 1998.
- [26] Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming*, pages 122–138. Springer, 1998.
- [27] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM (JACM)*, 63(1):1–67, 2016.
- [28] Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. Monotonic tpestates for the object calculus.
- [29] Mathias Jakobsen, Alice Ravier, and Ornela Dardha. Papaya: global tpestate analysis of aliased objects. In *Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming*, pages 1–13, 2021.
- [30] Pallavi Joshi and Koushik Sen. Predictive tpestate checking of multithreaded java programs. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 288–296. IEEE, 2008.
- [31] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.
- [32] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *ACM SIGPLAN Notices*, 50(1):637–650, 2015.

- [33] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. Accumulation analysis. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:30. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. ISSN: 1868-8969.
- [34] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. *CE Shannon and J. McCarthy*, 1951.
- [35] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Linear types for large-scale systems verification. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–28, 2022.
- [36] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. Path-sensitive and alias-aware tpestate analysis for detecting os bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–872, 2022.
- [37] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. Spata: Effective os bug detection with summary-based, alias-aware, and path-sensitive tpestate analysis. *ACM Transactions on Computer Systems*, 42(3-4):1–40, 2024.
- [38] Daniel Marino and Todd Millstein. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation*, pages 39–50, 2009.
- [39] Filipe Militão, Jonathan Aldrich, and Luís Caires. Aliasing control with view-based tpestate. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, pages 1–7, 2010.
- [40] Filipe Militão, Jonathan Aldrich, and Luís Caires. Substructural tpestates. In *Proceedings of the ACM SIGPLAN 2014 workshop on Programming languages meets program verification*, pages 15–26, 2014.
- [41] Peter O’Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019.
- [42] Dominic Orchard and Nobuko Yoshida. Using session types as an effect system. *arXiv preprint arXiv:1602.03591*, 2016.
- [43] Luca Padovani. Deadlock-free tpestate-oriented programming. *arXiv preprint arXiv:1803.10670*, 2018.
- [44] Rajshakhar Paul, Asif Kamal Turzo, and Amiangshu Bosu. Why security defects go unnoticed during code reviews? a case-control study of the chromium os project. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1373–1385. IEEE, 2021.

- [45] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th annual IEEE symposium on logic in computer science*, pages 55–74. IEEE, 2002.
- [46] Hannes Saffrich, Yuki Nishida, and Peter Thiemann. Law and order for typestate with borrowing. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024.
- [47] Hannes Saffrich and Peter Thiemann. Polymorphic typestate for session types. In *Proceedings of the 25th International Symposium on Principles and Practice of Declarative Programming*, pages 1–15, 2023.
- [48] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J Fink, and Eran Yahav. Alias analysis for object-oriented programs. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 196–232, 2013.
- [49] Robert E. Strom and Daniel M Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Transactions on Software Engineering*, 19(5):478–485, 2002.
- [50] Robert E Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE transactions on software engineering*, pages 157–171, 1986.
- [51] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and computation*, 111(2):245–296, 1994.
- [52] Eran Yahav and Stephen Fink. *The SAFE Experience*, pages 17–33. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.