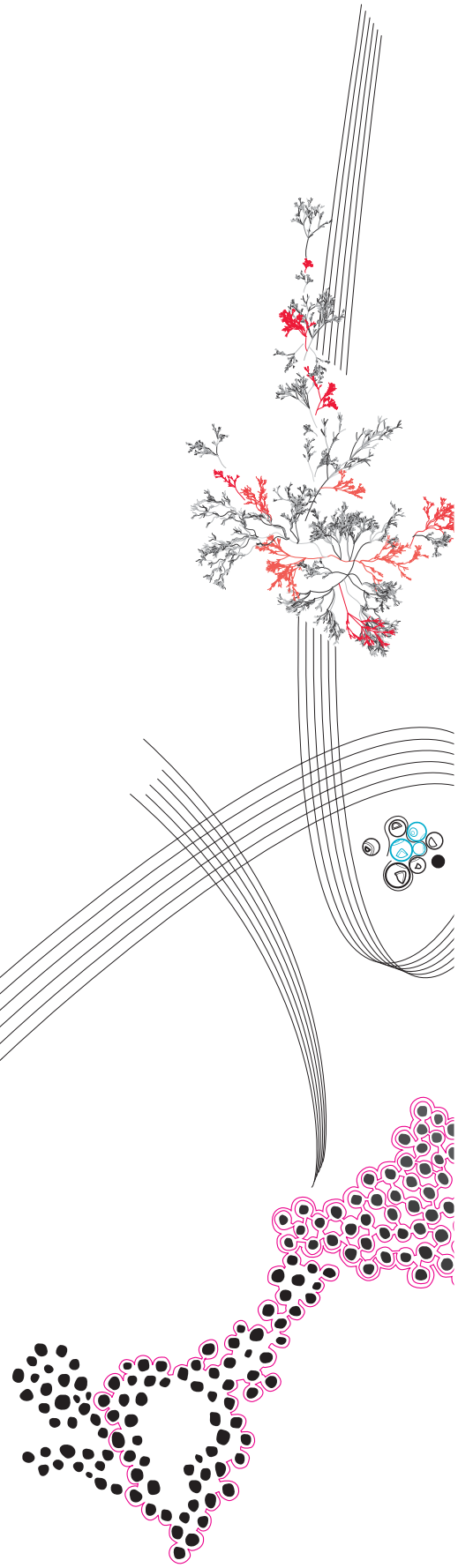

MSc Computer Science
Final Project

Practical Probabilistic Program Verification using Caesar

Franka van Jaarsveld

Committee:
dr.ir. A. Continella
prof.dr.ir. J.P. Katoen
S.M. Nicoletti MSc
P. Schröer, MSc
D. Haase, MSc

May , 2025

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

CONTENTS

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my daily supervisors, Stefano, Philipp, and Darion, for their expert guidance and unwavering support throughout this project, as well as for carefully reading and re-reading my thesis. Whenever I encountered obstacles, they helped me explore alternative approaches, and they consistently encouraged me to ask questions, even outside our weekly meetings, always responding promptly to my emails. Their support made this project feel like a shared priority, and I am deeply appreciative of their commitment.

I am also grateful to my friends and family for their constant support and encouragement throughout this process, and for asking about my thesis even when they had no idea what I was talking about.

AI DISCLAIMER

This thesis benefited from the use of AI writing tools, including OpenAI's ChatGPT, which was used to improve clarity and refine academic language. All content was reviewed and edited by the author, who takes full responsibility for the final version of the text.

Abstract

This thesis explores the practical verification of probabilistic programs using *Caesar*, a weakest preexpectation-based verification tool for reasoning about the expected behaviour of discrete probabilistic programs. The Bounded Retransmission Protocol (BRP) is studied as a case study. A key contribution of this work is the abstraction and decomposition of BRP into two geometric-like programs, enabling more effective reasoning about the protocol's behaviour and facilitating the stepwise verification strategy. Theoretical verification of key properties (positive almost-sure termination, success probability, and the expectation of the number of failed and sent transmissions) was largely successful using weakest preexpectation calculus. Translating these results into verification using Caesar introduced practical challenges, particularly in invariant discovery. Additionally, even valid invariants did not always lead to successful verification due to limitations in Caesar's current implementation, particularly in SMT solver performance and the handling of exponentials. However, through workarounds including alternative invariants and a 'fueled' exponential function, meaningful properties of BRP were verified. This thesis demonstrates how such techniques support practical verification in Caesar and concludes with a discussion of its strengths, limitations, and recommendations for effective use.

Keywords: Probabilistic programming, program verification, weakest preexpectation, weakest liberal preexpectation, expected runtime, fixed-point iteration, supremum, Caesar, loop-unrolling, induction, inductive invariants, HeyVL, Bounded Retransmission Protocol.

CHAPTER 1

INTRODUCTION

Probabilistic Programs

Probabilistic programs extend traditional programs by incorporating three constructs: the ability to draw values from distributions, the ability to make probabilistic choices, and the ability to condition values of variables via *observe* statements. These constructs enable the modelling of uncertainty within a program. Many established programming languages have probabilistic counterparts designed to support these features. Examples include PyMC for Python [1], WebPPL for Javascript [2], and STAN for C++ [3], among others.

Probabilistic programs are widely employed to represent probabilistic models, which are fundamental to numerous machine learning applications. These include, but are not limited to, information extraction, speech recognition, computer vision, coding theory, and reachability analysis [4]. Two remarkable examples highlight the practical importance of probabilistic programming. First, the seismic monitoring NET-VISA [5] was developed and deployed by the United Nations to enhance the International Monitoring System (IMS), a global sensor network developed for the Comprehensive Nuclear Test Ban Treaty. NET-VISA significantly improved event detection, reducing missed events by approximately 60% compared with the previous system, and successfully identified events overlooked by human analysts. Second, the probabilistic programming language Scenic [6] has been employed to generate specialised training and test sets for neural networks. In a case study focused on vehicle detection in road images, Scenic produced data that improved the performance of a convolutional neural network beyond what was achieved using state-of-the-art synthetic data generation methods. This application is particularly relevant for the development of autonomous vehicles, where accurate object detection is critical for safety and reliability.

Driven by such impactful applications, the use of probabilistic programs has grown significantly in recent years, accompanied by a corresponding increase in research on the verification of such programs [7].

Probabilistic Program Verification

Ensuring the correctness of software systems is critical is modern society, particularly for the software controlling transportation and communication infrastructure [8]. The failure of such systems can have severe consequences, as highlighted by several remarkable incidents. One prominent example is the 1992 failure of the London Ambulance Service's computer-aided dispatch system. Following its deployment, the system experienced significant malfunctions, resulting in significant delays in ambulance dispatching, with reports of up to 11-hour waits. Media sources at the time claimed that up to 30 people may have died as a result [9]. A more recent incident occurred in 2008, involving the Gamma Knife, a high-precision radiation therapy device used to treat brain diseases. A software bug

caused the emergency stop button to fail during a procedure, forcing medical personnel to manually remove the patient from the machine. Fortunately, no injuries occurred, but the incident highlights the potential dangers of software errors in critical systems [10].

As Dijkstra famously stated, "though testing a program is a very effective way to show the presence of bugs, it is hopelessly inadequate for showing their absence" [11]. To provide stronger assurances on the reliability of software, formal program verification aims to mathematically prove that a program behaves according to its specification. As early as 1949, Alan Turing outlined a method for the systematic verification of software and anticipated the development of calculi for such analysis [12]. In 1975, Dijkstra introduced one such formalism: the *weakest precondition calculus*, which allows reasoning about whether a program satisfies a given postcondition [13]. Building on this foundation, McIver and Morgan developed the *weakest preexpectation calculus*, a probabilistic extension of Dijkstra's calculus. This method enables reasoning about expected behaviours in probabilistic programs and systems [7].

Despite these advances, program verification remains a complex and labour-intensive task, requiring substantial expertise and manual effort. Verification tools have therefore become an essential aid, improving the efficiency and reliability of the verification process [14]. Reasoning about the expected behaviour of probabilistic programs is known to be strictly harder than for ordinary programs, but techniques developed to verify such programs often feature little to no automation. To address this, Schröer et al. introduced *Caesar*, a verification infrastructure based on weakest preexpectation theory, which facilitates reasoning about the expected behaviour of discrete probabilistic programs [15].

To evaluate the effectiveness of Caesar, its developers have applied it to various verification problems drawn from the literature [15]. One such case study is the *Bounded Retransmission Protocol* (BRP), a protocol that extends the well-known Alternating Bit Protocol. Unlike its predecessor, BRP does not rely on the fairness of data transmission channels, and limits the number of retransmissions to prevent non-termination [14]. Although conceptually simple, BRP exhibits non-trivial behaviour due to its combination of bounded retries and probabilistic failure, which complicates reasoning about expected outcomes and makes it an interesting candidate for verification studies [16].

Typical questions in probabilistic program analysis concern quantifying aspects of their expected behaviour, such as expected runtimes, expected values of program variables, and probabilities of certain outcomes [15]. Currently, Caesar has only been used to verify the expected number of failed transmissions for BRP under the constraint that the number of data frames is limited to three.

Research Goal

This study will generalise the existing verification of Caesar using BRP to support an arbitrary number of packets. In addition, it explores the verification of further properties concerning the global behaviour of BRP. Through this analysis, this study will address each of the central questions in probabilistic program analysis and provide deeper insights into Caesar's verification capabilities. Based on this assessment, the study aims to provide concrete recommendations for the effective use of Caesar.

To this end, a simplified representation of BRP is developed to support verification using the weakest preexpectation calculus. Using this model, a set of properties of BRP is identified for verification, and these are verified manually, thereby establishing a theoretical foundation for the Caesar verification.

The manual calculations are then used to generate Caesar programs for the practical verification of each property. This process is documented in detail, highlighting which

properties can be verified with Caesar, what limitations are encountered, and what insights are gained. Based on this experience, the study formulates a set of recommendations for using Caesar.

Research Questions

The central research question guiding this study is:

> **What are the advantages and shortcomings of Caesar's verification capabilities when applied to the Bounded Retransmission Protocol (BRP), and what recommendations for its application can be derived from this case study?**

To address this overarching question, the following sub-questions are examined:

1. How can BRP be modelled to facilitate the verification using Caesar?
2. Which properties of BRP can be verified using Caesar's underlying weakest preexpectation calculus?
3. How can theoretical verification based on weakest preexpectations be translated into verification using Caesar?
4. What challenges and insights emerge when translating the theoretical verification to practical verification using Caesar?
5. What are the verification results, and what assumptions, if any, are required to verify each property?

Thesis Structure

To address the research question outlined above, Chapter 2 introduces the necessary background on probabilistic programs, weakest preexpectations, Caesar, and the Bounded Retransmission Protocol. Chapter 3 presents a review of related literature. The methodology is described in Chapter 4, which includes the abstraction of BRP to a model that captures only its external behaviour, its subsequent refinement, and the selection of verification properties. Chapter 5 details the theoretical verification of the selected properties using manual weakest preexpectation calculations. These results form the foundation of the practical verification using Caesar, which is presented in Chapter 6. Chapter 7 discusses the outcomes of the verification efforts and reflects on the research questions. The thesis is concluded in Chapter 8, and potential directions for future work are outlined in Chapter 9.

CHAPTER 2

BACKGROUND

2.1 Probabilistic Programs

Probabilistic programs offer a few added constructs compared to regular programs: the ability to draw values from distributions, make a probabilistic choice, and condition values of variables using *observe* statements [4]. These features allow probabilistic programs to model uncertainty and randomness, making them valuable tools in domains such as statistics and machine learning. Applications include information extraction, computer vision, speech recognition, coding theory, biology, and reliability analysis.

A simple probabilistic program is illustrated in Listing 2.1:

Listing 2.1: A simple probabilistic program.

```
1 count := 0;
2 stop := false;
3
4 while (!stop) {
5     count := count + 1;
6     stop := true [0.5] stop := false;
7 }
8
9 return count;
```

This program computes the number of iterations of the `while` loop (stored in `count`). In each loop iteration, a fair coin flip determines whether the program exits or continues. Though this example is straightforward, writing probabilistic programs is notoriously challenging, and reasoning about their expected behaviour is known to be strictly harder than ordinary programs [17]. Typical questions in probabilistic program analysis concern expected runtimes, expected values of program variables (e.g. the expected value of `count`), and conditional expected values and probabilities (e.g. the probability that `count` $= 1$).

The example program above is written in the *probabilistic guarded command language* (pGCL), originally introduced by McIver and Morgan [7] as an extension of Dijkstra's guarded command language (GCL) [18] to support probabilistic programs. This section provides a formal definition of pGCL, as presented by Kaminski [12].

Definition 1 (The Probabilistic Guarded Command Language).

(A) The set of **program states** is defined as

$$\Sigma = \{\sigma | \sigma : \text{Vars} \to \text{Vals}\}$$

where Vars represents a countable set of program variables, and Vals denotes a countable set of values. Unless otherwise specified, Vals $= \mathbb{Q}$, where $\mathbb{Q}$ represents the set of rational numbers.

(B) A **probability distribution** over values is a function $\pi : \text{Vals} \to [0, 1]$ for which

$$\sum_{v \in \text{Vals}} \pi(v) = 1$$

The set of all probability distributions over values is denoted by $\mathcal{D}(\text{Vals})$. A *distribution expression* is a function

$$\mu : \Sigma \to \mathcal{D}(\text{Vals})$$

that maps every program state to a probability distribution over values. This probability distribution is *discrete*, as Vals is countable by definition.

(C) The set of **programs** in pGCL is defined by the following grammar:

$$\begin{array}{lll} C \to & \texttt{skip} & \text{(effectless program)} \\ & \mid \texttt{diverge} & \text{(freeze)} \\ & \mid x := E & \text{(assignment)} \\ & \mid x :\approx \mu & \text{(random assignment)} \\ & \mid C; C & \text{(sequential composition)} \\ & \mid \texttt{if}(\varphi)\{C\}\texttt{else}\{C\} & \text{(conditional choice)} \\ & \mid \{C\}[p]\{C\} & \text{(probabilistic choice)} \\ & \mid \texttt{while}(\varphi)\{C\} & \text{(while loop)} \end{array}$$

Where $x \in \text{Vars}$ denotes a program variable, E represents an arithmetic expression over program variables, μ a distribution expression, φ a Boolean expression, and p a probability expression.

(D) A pGCL program containing no `diverge` statements or while loops is called **loop-free**.

(E) For a program state σ and arithmetic expression E, the notation $\sigma(E)$ denotes the **evaluation** of E in σ. This corresponds to substituting every occurrence of a program variable x in E by $\sigma(x)$. Similarly, $\sigma(\varphi)$ evaluates a Boolean expression φ in σ, yielding either *true* or *false*. For a value $v \in \text{Vals}$, the expression $\sigma[x := v]$ represents a modified program state in which the variable x is updated to v, while all other variables remain unchanged.

(F) The final piece of required notation involves **Iverson brackets**. The Iverson bracket $[\varphi]$ of guard φ is defined as the following function:

$$[\varphi] : \Sigma \to \{0, 1\}, [\varphi](\sigma) = \begin{cases} 1 & \text{if } \sigma(\varphi) = \mathit{true}, \\ 0 & \text{if } \sigma(\varphi) = \mathit{false}. \end{cases}$$

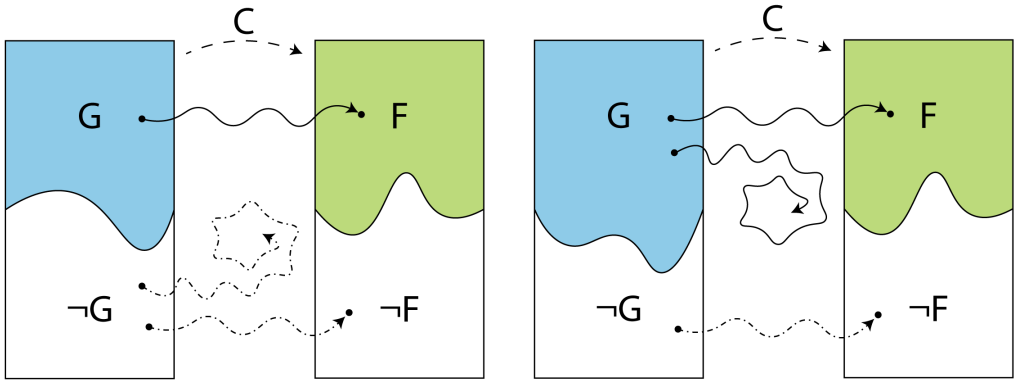

FIGURE 2.1: Graphical representation of weakest preconditions (left) and weakest *liberal* preconditions (right), given a program C, precondition G and postcondition F. Diverging runs, represented by spiralling arrows, indicate instances where the program fails to terminate.

2.2 Weakest Preexpectations

The analysis of probabilistic programs can be approached using *weakest preexpectations*, a formal reasoning framework developed by McIver & Morgan [7]. This framework builds upon Dijkstra's *weakest precondition* calculus for non-probabilistic programs [13], and understanding the latter is essential in grasping the former. Accordingly, this section begins with an introduction to the general concept of weakest preconditions, followed by the presentation and formal definition of weakest preexpectations, along with the necessary domain theory. Subsequently, bounded expectations and weakest liberal preexpectations are introduced and formally defined, followed by a section on reasoning about loops. Finally, two levels of probabilistic termination are defined, and the expected runtime calculus is introduced and formally defined. This section is primarily based on the work of Kaminski, with notations and definitions adopted from his work [12], unless explicitly stated otherwise.

Consider a program C whose behaviour needs verification and an accompanying *specification* that gives a postcondition F on the final states that C may reach. The aim is to prove that the program C will only terminate in states that satisfy F, thus meeting its specification. This can be approached by calculating the largest set of initial states that guarantee termination in a state satisfying F. This set of states is the *weakest precondition* G. Figure 2.1 (left) illustrates weakest precondition reasoning. Diverging runs of C, depicted as a spiralling arrow, are excluded from G since they failed to guarantee termination in a state satisfying F.

When verifying a probabilistic program, the chosen postcondition is usually an *expectation*, expressed either in the form of $[F]$ or as a variable x. In the former case, the *weakest preexpectation* framework effectively obtains a guarantee on the probability that the program terminates in a state satisfying F. In the latter case, the weakest preexpectation yields the expected value of x when executing the program from an initial state σ.

Before formally defining an *expectation*, we must first define *partial orders* and *complete lattices*:

C	$\mathbf{wp}[\![C]\!](f)$
`skip`	f
`diverge`	$\mathbf{0}$
$x := E$	$f[x := E]$
$x :\approx \mu$	$\lambda\sigma. \int_{\text{Vals}} (\lambda v. f(\sigma[x := v]))d\mu(\sigma)$
$P; Q$	$\text{wp}[\![P]\!](\text{wp}[\![Q]\!](f))$
$\texttt{if}(\varphi)\{P\}\texttt{else}\{Q\}$	$[\varphi] \cdot \text{wp}[\![P]\!](f) + [\neg\varphi] \cdot \text{wp}[\![Q]\!](f)$
$\{P\}[p]\{Q\}$	$p \cdot \text{wp}[\![P]\!](f) + (1-p) \cdot \text{wp}[\![Q]\!](f)$
$\texttt{while}(\varphi)\{P\}$	$\text{lfp } X.([\varphi] \cdot \text{wp}[\![P]\!](X) + [\neg\varphi] \cdot f)$

TABLE 2.1: Weakest preexpectation calculus.

Definition 2 (Partial Orders). [19]

(A) Let D be some universe. Then $(D, \sqsubseteq)$, where $\sqsubseteq$ is a binary relation $\sqsubseteq \subseteq D \times D$, is a **partial order**, iff $\sqsubseteq$ is

(a) reflexive, i.e. for all $a \in D$

$$a \sqsubseteq a,$$

(b) transitive, i.e. for all $a, b, c \in D$

$$a \sqsubseteq b \text{ and } b \sqsubseteq c \text{ implies } a \sqsubseteq c,$$

(c) and antisymmetric, i.e. for all $a, b \in D$

$$a \sqsubseteq b \text{ and } b \sqsubseteq a \text{ implies } a = b.$$

Whenever the universe D is evident from the context, we may omit the D from $(D, \sqsubseteq)$ and simply speak of the partial order $\sqsubseteq$.

(B) An element $d \in D$ is called an **upper bound** of $S \sqsubseteq D$, denoted $S \sqsubseteq d$, if for every $s \in S$, $s \sqsubseteq d$. An upper bound d of $S \sqsubseteq D$ is called a least upper bound, or **supremum** of S if

$$d \sqsubseteq d' \text{ for every upper bound } d' \text{ of } S.$$

(C) An element $d \in D$ is called a **lower bound** of $S \sqsubseteq D$, denoted $d \sqsubseteq S$, if for every $s \in S$, $d \sqsubseteq s$. A lower bound d of $S \sqsubseteq D$ is called a greatest lower bound, or **infimum** of S if

$$d' \sqsubseteq d \text{ for every lower bound } d' \text{ of } S.$$

(D) $S \sqsubseteq D$ is called a **chain** in D if, for every $s_1, s_2 \in S$,

$$s \sqsubseteq s_2 \text{ or } s_2 \sqsubseteq s_1.$$

Definition 3 (Complete Lattices). [19]
A partial order $(D, \sqsubseteq)$ is called a complete lattice if every subset $S \subseteq D$ has a supremum $\sup S \in D$ and an infimum $\inf S \in D$.
Note that every complete lattice $(D, \sqsubseteq)$ has a least element $\bot$ and dually a greatest element $\top$ which satisfy

$$\forall a \in D : \quad \bot \sqsubseteq a \sqsubseteq \top.$$

Definition 4 (Expectations).
The set of expectations $\mathbb{E}$ is defined as

$$\mathbb{E} = \{f | f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}\}$$

where $(\mathbb{E}, \sqsubseteq)$ forms a complete lattice with the partial order:

$$g \sqsubseteq h \iff \forall \sigma \in \Sigma . g(\sigma) \leq h(\sigma).$$

The least element of $(\mathbb{E}, \sqsubseteq)$ is the constant function $\lambda s.0$, denoted $\mathbf{0}$. Analogously, its greatest element is denoted by $\mathbf{1}$. The supremum of a subset $S \subseteq \mathbb{E}$ is constructed point-wise as:

$$\sup S = \lambda \sigma . \sup_{f \in S} f(\sigma).$$

Given a postexpectation $f \in \mathbb{E}$ and a probabilistic program C, the weakest preexpectation $g \in \mathbb{E}$ is a function that maps each initial state σ to the expected value of f after the execution of C on input σ. This function, denoted $\mathrm{wp}[\![C]\!](f) : \mathbb{E} \to \mathbb{E}$, is systematically derived using the weakest preexpectation calculus outlined in Table 2.1. As shown in the table, the weakest preexpectation of a while loop is expressed as the least fixed point of the loop characteristic function $\Phi_f(X)$:

$$\mathrm{wp}[\![\texttt{while}(\varphi)\{P\}]\!](f) = \mathrm{lfp}\ X. \underbrace{([\varphi] \cdot \mathrm{wp}[\![P]\!](X) + [\neg\varphi] \cdot f)}_{\Phi_f(X)}.$$

To further simplify this expression, Kleene's Fixed Point Theorem may be applied. The theorem is presented below, preceded by several preliminary notions necessary for its comprehension.

Definition 5 (Continuity). [19]
Let $(D, \sqsubseteq)$ be a complete lattice and let $\Phi : D \to D$. Then Φ is called continuous, iff for every chain $S = \{s_0 \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq ...\} \subseteq D$

$$\Phi(\sup S) = \sup \Phi(S),$$

where $\Phi(S)$ is the standard shorthand for the set $\{\Phi(a) | a \in S\}$.

Definition 6 (Monotonicity). [19]
Let $(D, \sqsubseteq)$ be a complete lattice and let $\Phi : D \to D$. Then Φ is called monotonic, iff for all $a, b \in D$

$$a \sqsubseteq b \text{ implies } \Phi(a) \sqsubseteq \Phi(b).$$

Theorem 1 (Continuity implies Monotonicity).
Every continuous function is monotonic. For a formal proof, refer to [12].

Theorem 2 (Kleene Fixed Point Theorem). [19, 20]
Let $(D, \sqsubseteq)$ be a complete lattice with a least element $\bot$ and greatest element $\top$. Moreover, let $\Phi : D \to D$ be continuous (thus monotonic). Then Φ has a least fixed point lfp Φ and a greatest fixed point gfp Φ, respectively given by

$$\text{lfp } \Phi = \sup_{n\in\mathbb{N}} \Phi^n(\bot) \text{ and gfp } \Phi = \inf_{n\in\mathbb{N}} \Phi^n(\top).$$

By Kleene's Fixed Point Theorem, it follows that:

$$\text{wp}[\![\texttt{while}(\varphi)\{P\}]\!] = \text{lfp } X.\Phi_f(X) = \sup_{n\in\mathbb{N}} \Phi_f^n(\mathbf{0})$$

where Φ_f^n represents the n-fold application of Φ_f and the supremum (sup) corresponds to the least upper bound (LUB).

2.2.1 Weakest Liberal Preexpectations

Weakest preconditions consider only initial states that lead to final states satisfying the postcondition, discarding diverging runs. However, in cases where the postcondition must hold *if* the program terminates, reasoning is conducted using weakest *liberal* preconditions. This concept is illustrated in the right half of Figure 2.1. Similarly, weakest preexpectations also exclude diverging runs. However, when the objective is to determine the probability that a program terminates in a desired state *or* diverges, reasoning is conducted using weakest liberal preexpectations. In contrast to weakest preexpectations, which reason about total correctness, weakest liberal preexpectations do so about partial correctness. Furthermore, weakest liberal preexpectations specifically address *one-bounded* expectations:

Definition 7 (Bounded Expectations).
A set of bounded expectations is defined as:

$$\mathbb{E}_{\leq \exists b} = \{f \in \mathbb{E} | \exists b \in \mathbb{R}_{\geq 0} : f \sqsubseteq b\}$$

where the order relation $f \sqsubseteq g$ is given by

$$f \sqsubseteq g \iff \forall \sigma \in \Sigma : f(\sigma) \leq g(\sigma).$$

Although $(\mathbb{E}_{\leq \exists b}, \sqsubseteq)$ forms a lattice with least element $\mathbf{0}$, it is not complete, as a supremum may not exist. Consequently, Kleene's Fixed Point theorem does not guarantee the existence of least fixed points for bounded expectations.

C	$\mathbf{wlp}[\![C]\!](f)$
`skip`	f
`diverge`	$\mathbf{1}$
$x := E$	$f[x := E]$
$x :\approx \mu$	$\lambda\sigma. \int_{\text{Vals}} (\lambda v. f(\sigma[x := v]))d\mu(\sigma)$
$P; Q$	$\text{wlp}[\![P]\!](\text{wlp}[\![Q]\!](f))$
$\texttt{if}(\varphi)\{P\}\texttt{else}\{Q\}$	$[\varphi] \cdot \text{wlp}[\![P]\!](f) + [\neg\varphi] \cdot \text{wlp}[\![Q]\!](f)$
$\{P\}[p]\{Q\}$	$p \cdot \text{wlp}[\![P]\!](f) + (1-p) \cdot \text{wlp}[\![Q]\!](f)$
$\texttt{while}(\varphi)\{P\}$	$\text{gfp } X.([\varphi] \cdot \text{wlp}[\![P]\!](X) + [\neg\varphi] \cdot f)$

TABLE 2.2: Weakest liberal preexpectation calculus. The differences with Table 2.1 are highlighted in blue.

Definition 8 (One-bounded Expectations).
A set of one-bounded expectations, denoted $\mathbb{E}_{\leq 1}$ is defined as

$$\mathbb{E}_{\leq 1} = \{f \in \mathbb{E} | f \sqsubseteq 1\}.$$

$(\mathbb{E}_{\leq 1}, \sqsubseteq)$ is a complete lattice with least element $\mathbf{0}$ and greatest element $\mathbf{1}$. Suprema and infima are constructed as in $\mathbb{E}$ (Definition 4).

For a predicate F and a probabilistic program C, the weakest preexpectation $g \in \mathbb{E}_{\leq 1}$ with respect to the postexpectation $[F] \in \mathbb{E}_{\leq 1}$ is an expectation such that $g(\sigma)$ represents the probability that executing C on input σ either terminates or ends in a state $\tau \models F$. This function is denoted $\text{wlp}[\![C]\!](f) : \mathbb{E}_{\leq 1} \to \mathbb{E}_{\leq 1}$, and the corresponding calculus is provided in Table 2.2. As shown in the table, the weakest liberal preexpectation of a while loop is defined as the *greatest fixed point* of the loop characteristic function $\Phi_f(X)$:

$$\text{wlp}[\![\texttt{while}(\varphi)\{P\}]\!](f) = \text{gfp } X. \underbrace{([\varphi] \cdot \text{wlp}[\![P]\!](X) + [\neg\varphi] \cdot f)}_{\Phi_f(X)}.$$

By Kleene's Fixed Point Theorem (Theorem 2), it follows that:

$$\text{gfp } X.\Phi_f(X) = \text{inf}_{n\in\mathbb{N}}\Phi_f^n(\mathbf{1})$$

where Φ_f^n denotes the n-fold application of Φ_f and the infimum (inf) corresponds to the greatest lower bound (GLB).

A comparison of the graphical representations of weakest preexpectations and weakest liberal preexpectations reveals that wp- and wlp-calculus are equivalent for non-diverging programs. Formally, the relationship between wp- and wlp-calculus is expressed as:

Theorem 3 (Relationship wp and wlp).
Let $C \in \text{pGCL}$ and $f \in \mathbb{E}_{\leq 1}$. Then

$$\text{wlp}[\![C]\!](f) = \text{wp}[\![C]\!](f) + \mathbf{1} - \text{wp}[\![C]\!](\mathbf{1}).$$

Additionally, let C terminate almost-surely, i.e. let $\text{wp}[\![C]\!](\mathbf{1}) = \mathbf{1}$ (see Section 2.2.4). Then

$$\text{wlp}[\![C]\!](f) = \text{wp}[\![C]\!](f).$$

2.2.2 Reasoning About Loops

While verifying loop-free programs is relatively straightforward, and weakest preexpectations can typically be computed in practice, reasoning about while-loops remains one of the most challenging aspects of program verification [12]. In some cases, the property of interest may be undecidable using weakest preexpectation reasoning. Fortunately, many correctness properties can be expressed as either upper or lower bounds on preexpectations. This section explores how induction and co-induction can be employed to compute these bounds for programs containing loops.

Invariants

A key concept to understanding the use of induction and co-induction in this context is the notion of *invariants*. Additionally, an important prerequisite for comprehending invariants is the concept of *Hoare triples*:

Definition 9 (Hoare Triples). [21, 22]
Given two predicated F and G and a program C, a Hoare triple $\langle G \rangle C \langle F \rangle$ is said to be *valid* iff

If the program C started in some initial state $\sigma \models G$,
then C terminates in some final state $\tau \models F$.

Broadly, an invariant captures the behaviour of a loop. For a non-probabilistic program `while`$(\varphi)\{C\}$, any I satisfying the Hoare triple $\langle \varphi \wedge I \rangle C \langle I \rangle$ is called a *loop invariant*. For probabilistic programs, the definition of a loop invariant is more nuanced, requiring a distinction between superinvariants and subinvariants.

Definition 10 (Invariants).
Let Φ_f denote the wp-loop characteristic function of $\texttt{while}(\varphi)\{C\}$ with respect to postexpectation $f \in \mathbb{E}$, and let $I \in \mathbb{E}$. Then:

- I is a wp-*superinvariant* of $\texttt{while}(\varphi)\{C\}$ with respect to postexpectation f iff:

$$\Phi_f(I) \sqsubseteq I,$$

- I is a wp-*subinvariant* of $\texttt{while}(\varphi)\{C\}$ with respect to postexpectation f iff:

$$I \sqsubseteq \Phi_f(I).$$

Super- and subinvariants can similarly be defined for wlp-characteristic functions, although these invariants and f belong to the type $\mathbb{E}_{\leq 1}$ rather than $\mathbb{E}$.

Induction

Induction on natural numbers is a well-established proof principle stating that to prove that a predicate F holds for all natural numbers, it suffices to prove the following:

1. $0 \models F$, and
2. $n \models F$ implies $n + 1 \models F$.

This principle can be extended to continuous functions on complete lattices, resulting in *Park's Lemma*:

Lemma 1 (Park's Lemma). [23]
Let $(D, \sqsubseteq)$ be a complete lattice, $d \in D$, and $\Phi : D \to D$ a monotonic function. Then:

$$\Phi(d) \sqsubseteq d \implies \text{lfp } \Phi \sqsubseteq d,$$

and dually:

$$d \sqsubseteq \Phi(d) \implies d \sqsubseteq \text{gfp } \Phi.$$

Since weakest preexpectations are defined as least fixed points of continuous functions on complete lattices, Park's Lemma enables reasoning about the upper bounds of weakest preexpectations, and dually, the lower bounds of weakest liberal preexpectations. Specifically:

Theorem 4 (Induction for Upper Bounds on wp).
If $I \in \mathbb{E}$ is a wp-superinvariant of $\texttt{while}(\varphi)\{C\}$ with respect to postexpectation f, then:

$$\text{wp}[\![\texttt{while}(\varphi)\{C\}]\!](f) \sqsubseteq I.$$

Theorem 5 (Coinduction for Lower Bounds on wlp).
If $I \in \mathbb{E}_{\leq 1}$ is a wlp-subinvariant of $\texttt{while}(\varphi)\{C\}$ with respect to postexpectation f, then:

$$I \sqsubseteq \text{wlp}[\![\texttt{while}(\varphi)\{C\}]\!](f).$$

While induction allows reasoning about an *upper bound* of the least fixed point and coinduction enables reasoning about a *lower bound* of the greatest fixed point, it is not possible to reason about a *lower* bound of a least fixed point or an *upper* bound of a greatest fixed point using the same approach. Kaminski [12] discusses alternative proof rules for obtaining such bounds. However, these rules, which involve the use of ω-invariants, are less elegant and more complex to apply. For these alternative methods, refer to Chapter 5.2.4 of [12].

2.2.3 Expected Runtimes

Reasoning about expected runtimes of probabilistic programs is surprisingly subtle and full of nuances, which underlines the desire for formal methods suited for reasoning about expected runtimes [24]. In the context of probabilistic programs, a runtime is defined as a function that maps initial program states to non–negative real numbers or infinity, representing the expected execution time. Formally, runtimes are defined as follows:

Definition 11 (Runtimes).
The set of runtimes $\mathbb{T}$ is defined to coincide with the set of expectations (see Definition 4):

$$\mathbb{T} = \{t | t : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}\} = \mathbb{E}.$$

Consequently, the complete lattice $(\mathbb{T}, \sqsubseteq)$, its least element, and the construction of suprema are defined exactly as for expectations, i.e. the order relation is given by:

$$s \sqsubseteq t \iff \forall \sigma \in \Sigma. s(\sigma) \sqsubseteq t(\sigma);$$

the least element is $\lambda\sigma.0$, denoted by $\mathbf{0}$; and the supremum of a subset $S \subseteq \mathbb{T}$ is constructed point-wise as:

$$\sup S = \lambda\sigma. \sup_{f \in S} f(\sigma).$$

Given a postruntime $t \in \mathbb{T}$ and a probabilistic program C, the preruntime $s \in \mathbb{T}$ is a function that maps each initial state σ to the expected time required to execute C on σ, including the additional time t that elapses after C terminates. This function, denoted $\text{ert}[\![C]\!](t) : \mathbb{T} \to \mathbb{T}$, is systematically derived using the expected runtime calculus presented in Table 2.3. Analogous to the weakest preexpectation calculus, the expected runtime of a while-loop is defined as the least fixed point (or, equivalently, the supremum, as per Theorem 2) of the loop characteristic function:

$$\text{ert}[\![\texttt{while}(\varphi)\{P\}]\!](t) = \text{lfp } X. \underbrace{(\mathbf{1} + [\varphi] \cdot \text{ert}[\![P]\!](X) + [\neg\varphi] \cdot t)}_{\Phi_t(X)} = \sup_{n \in \mathbb{N}} \Phi_t^n(\mathbf{0}).$$

An intuitive alternative approach to determining a program's expected runtime involves using the weakest preexpectation calculus with user-defined reward structures. However, this approach is not equivalent to the ert-calculus. To illustrate this distinction, consider

C	$\mathbf{ert}[\![C]\!](t)$
`skip`	$\mathbf{1} + t$
`diverge`	∞
$x := E$	$\mathbf{1} + t[x := E]$
$x :\approx \mu$	$1 + \lambda\sigma. \int_{\text{Vals}} (\lambda v.t(\sigma[x := v]))d\mu(\sigma)$
$P; Q$	$\text{ert}[\![P]\!](\text{ert}[\![Q]\!](t))$
$\texttt{if}(\varphi)\{P\}\texttt{else}\{Q\}$	$\mathbf{1} + [\varphi] \cdot \text{ert}[\![P]\!](t) + [\neg\varphi] \cdot \text{ert}[\![Q]\!](t)$
$\{P\}[p]\{Q\}$	$\mathbf{1} + p \cdot \text{ert}[\![P]\!](t) + (1-p) \cdot \text{ert}[\![Q]\!](t)$
$\texttt{while}(\varphi)\{P\}$	$\text{lfp } X.(1 + [\varphi] \cdot \text{ert}[\![P]\!](X) + [\neg\varphi] \cdot t)$

TABLE 2.3: Expected runtime calculus.

an extension of the pGCL grammar with the statement `reward (r)`, where $r \in \mathbb{Q}^{\infty}_{\geq 0}$ represents a rational reward or ∞. The corresponding amendment to the wp-calculus is given by $\text{wp}[\![\texttt{reward (r)}]\!](f) = r + f$. As defined in [25], $\text{wp}[\![C]\!](X)$ then represents the expected reward obtained from executing program C on an initial state $\sigma \in \Sigma$ and collecting reward $X(\tau)$ *upon termination* in some state τ. Notably, this formulation assigns an expected reward of 0 to diverging programs, whereas the actual expected runtime in such cases is ∞. Therefore, the weakest preexpectation calculus with user-defined rewards cannot serve as a substitute for the ert-calculus.

2.2.4 Termination

Termination is a fundamental liveness property of probabilistic programs, but it is significantly more nuanced and subtle compared to nonprobabilistic programs [12]. Several levels of termination exist for probabilistic programs, two of which are relevant to this thesis:

Definition 12 (Probabilistic Termination).
Let C be a pGCL program. Then C terminates universally

(a) **almost-surely** $\iff \text{wp}[\![C]\!](\mathbf{1}) = \mathbf{1}$,

(b) **positively almost-surely** $\iff \forall\sigma \in \Sigma : \text{ert}[\![C]\!](\sigma) \sqsubset \infty$,
where ert represents the expected runtime of the program (see Table 2.3).

These classifications are ordered from weakest to strongest: a program that is positively almost-surely terminating is, by definition, also almost-surely terminating, although the inverse does not hold. For a more comprehensive explanation and more termination levels, refer to [12].

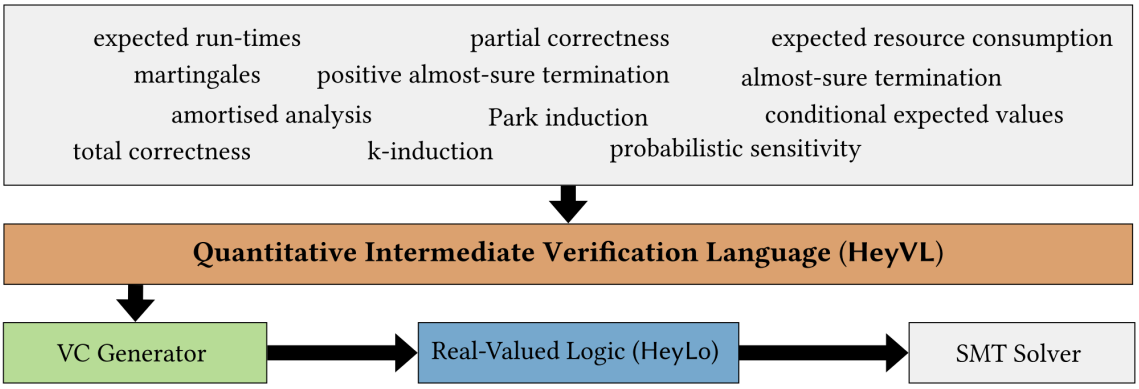

FIGURE 2.2: The architecture of the verification infrastructure Caesar [15].

2.3 Caesar

Many techniques exist for verifying probabilistic programs, but these often feature little (or no) automation [15]. Schröer et al. [15] introduced a novel verification infrastructure based on weakest preexpectation theory for reasoning about the expected behaviour of discrete probabilistic programs. Their tool, Caesar, implements this architecture, as shown in Figure 2.2.

The uppermost section of the figure (shaded in grey) displays complex verification techniques one might want to encode using the infrastructure. The inclusion of the quantitative intermediate verification language *HeyVL* (shaded in orange) enables the encoding of these techniques whilst also separating the back-end for efficient development.

A HeyVL program is a collection of *procedures*:

Definition 13 (HeyVL procedure). [15]
A HeyVL procedure `P` is structured as follows:

```
proc P (in : τ) -> (out : τ)
    pre ϕ
    post ψ
{S}
```

Where $\overline{in}$ and $\overline{out}$ represent lists of typed read-only inputs and outputs, and ϕ and ψ specify the preexpectation and postexpectation, respectively. The procedure body `S` is a HeyVL *statement*, whose syntax is given in [26].

A formal syntax of a HeyVL statement is given in [15]. However, as Caesar remains under active development, this does not fully reflect the current syntax. For the most up-to-date syntax of HeyVL, see [26]. In the context of this thesis, the relevant HeyVL constructs are variable declarations, procedure calls, boolean choices (i.e., if-statements), and while-loops. A procedure call passes parameters to a (co)procedure, executes its body, and assigns the result to the specified return variables.

A HeyVL procedure P verifies that the inequality $\phi \sqsubseteq \text{wp}[\![\texttt{S}]\!](\psi)$ holds, i.e. the expected value of ψ after the execution of `S` is lower-bounded by ϕ. This inequality is also called the *verification condition* (VC) of P (shaded in green in Figure 2.2). Upper bounds on expected values can be verified using *coprocedures*, which follow the same structure as procedures but replace `proc` by `coproc` in HeyVL, and $\sqsubseteq$ by $\sqsupseteq$ in their verification condition.

proof rule	`@wp`	`@wlp`	`@ert`
induction	overapproximation (`coproc`)	underapproximation (`proc`)	overapproximation (`coproc`)
loop unrolling	underapproximation (`proc`)	overapproximation (`coproc`)	underapproximation (`proc`)

TABLE 2.4: Soundness of the combinations of relevant proof rules and the supported calculus annotations in Caesar, taken from [26].

The generated verification conditions can be reduced to checking inequalities between *HeyLo* formulae (shaded in blue in Figure 2.2). HeyLo, a syntax for quantitative verification of probabilistic programs, aims to take the role that predicate logic has for classical verification [15]. The HeyLo inequalities are subsequently passed to an SMT (Satisfiability Modulo Theory) solver, which attempts to solve them.

In summary, Caesar accepts a HeyVL program consisting of procedures and coprocedures, generates verification conditions as HeyLo inequalities, and translates them into SMT queries. The translation to SMT may lead to undecidable theories, in which case Caesar returns *unknown*. Otherwise, Caesar verifies the program or provides a counterexample where verification conditions fail. For further details on Caesar, HeyVL and HeyLo, see [15].

2.3.1 Proof Rules for While Loops

As discussed in Section 2.2.2, reasoning about while-loops is one of the most challenging aspects of program verification. In Caesar, while-loops must be annotated with appropriate proof rules to enable verification. Caesar supports multiple proof rules, three of which are particularly relevant to this thesis and were introduced in the previous sections: induction, almost-sure termination, and positive almost-sure termination. To facilitate the correct application of these proof rules, Caesar provides annotations for (co)procedures that specify the desired calculus. The currently supported annotations are:

- `@wp`: weakest preexpectation,
- `@wlp`: weakest liberal preexpectation, and
- `@ert`: expected runtime.

When these annotations are incorporated, Caesar can automatically verify the soundness of the selected proof rules with respect to the specified calculus. Table 2.4 summarises the soundness relationships between calculus annotations and proof rules. Overapproximation is used when verifying upper bounds, whereas underapproximation is used to verify lower bounds.

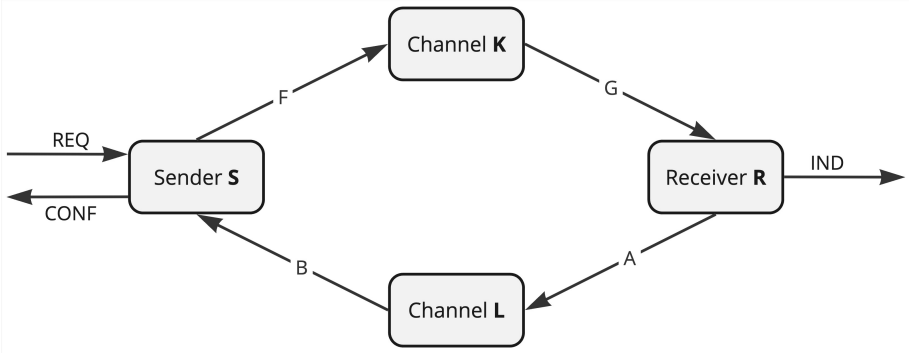

FIGURE 2.3: Overview of the Bounded Retransmission Protocol without the disconnect service, as presented in [14].

2.4 Bounded Retransmission Protocol

The Bounded Retransmission Protocol (BRP) is a communication protocol developed by Philips Electronics to communicate messages of arbitrary length over unreliable channels. In this protocol, messages are divided into smaller data frames with the following structure

Definition 14 (BRP data frame structure).

data frame = {
first $\in \{0, 1\}$: indicates whether this is the first frame,
last $\in \{0, 1\}$: indicates whether this is the last frame,
toggle $\in \{0, 1\}$: alternating bit to distinguish between subsequent frames,
datum: data to be transmitted, not included in acknowledgement frames
}

BRP can be seen as an extended version of the Alternating Bit Protocol, using a stop-and-wait approach known as 'positive acknowledgement with retransmission' (the sender waits for an acknowledgement before sending the next frame) [14]. If no acknowledgement is received, the sender retransmits the frame. However, the number of retransmissions is limited, so successful delivery is not guaranteed. Despite its simplicity, the behaviour of BRP is complex, making it a benchmark example in verification studies [16]. A model of BRP as presented in [14] is shown in Figure 2.3. The original protocol includes an additional disconnect service that allows the sender or receiver to disrupt the transmission, however, this has been omitted for simplicity. The model includes a sender (S), receiver (R), lossy communication channels (K) and (L) and the communication services, and its services are defined in Definition 15.

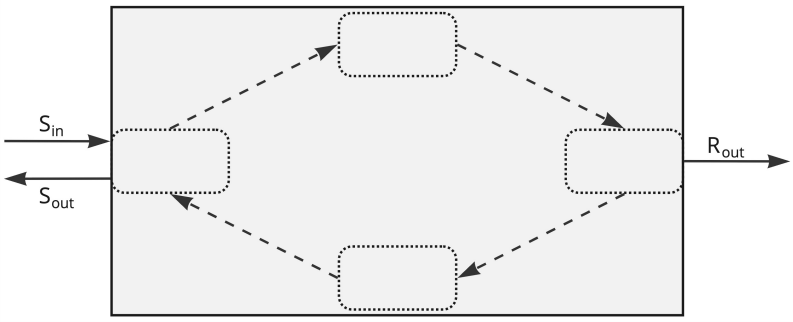

FIGURE 2.4: A simplified representation of the Bounded Retransmission Protocol, as presented in [27].

Definition 15 (BRP protocol services). [14]
The services included in the BRP protocol as illustrated in Figure 2.3. A data frame with index k is represented by d_k, and *ack* is an acknowledgement frame.
REQ: $s = [d_1, .., d_n]$ for $n > 0$
CONF: $c \in$ {C_OK, C_NOT_OK, C_DONT_KNOW}

C_OK: The request has been dispatched successfully

C_NOT_OK: The request has not been dispatched completely

C_DONT_KNOW: The request may or may not have been handled completely. This occurs when the last frame is sent but not acknowledged.

IND: (d_k, i)
where $i \in$ {I_FIRST, I_INCOMPLETE, I_OK, I_NOT_OK}, $0 < k \leq n$

I_FIRST: This packet is the first one

I_INCOMPLETE: This packet is an intermediate one; there is more data to come

I_OK: This was the final packet

I_NOT_OK: Contact with the server has been lost.

F, G: d_k where $0 < k \leq n$
A, B: *ack*

2.4.1 Abstraction for Verification

As mentioned in the previous section, the Bounded Retransmission Protocol (BRP) has been the subject of numerous verification studies, which will be examined in Section 3. These studies often employ simplified models of BRP that are tailored to the specific requirements of their verification objectives. Similarly, this study adopts a simplified model of BRP in which the sender, receiver, and communication channels are treated as a "black box". Only the inputs and outputs of the protocol are considered, allowing the focus to be placed on the practical effect of the protocol. This approach is illustrated in Figure 2.4, adapted from [27], where S_{in} corresponds to REQ, S_{out} to CONF, and R_{out} to IND. Additionally, the following probabilistic program, taken from [28], provides another simplified

representation of BRP:

Listing 2.2: A simplified model of BRP, as presented in [28].

```
sent := 0;
count := 0;

while count < MAX ∧ sent < N do
    first := 1 [p] first := 0;
    if first = 1 then
        second := 1 [q] second := 0;
        if second = 1 then
            sent := sent + 1;
            count := 0;
        else
            count := count + 1;
    else
        count := count + 1
state := sent = N;
return state
```

In this program, `p` represents the probability of successful packet transmission, `q` the probability of a successful acknowledgement, `MAX` the maximum number of consecutive failures without successful transmission, and `N` the total number of packets to be transmitted. The variable `sent` tracks the number of successfully transmitted data frames, while the variable `count` counts the number of consecutive failures. Both variables are essential for the loop condition.

Both models offer a simplified view of BRP, facilitating verification studies that focus primarily on its external behaviour. Although this abstraction results in the omission of certain details concerning the program's internal mechanisms, these details may not be essential for the verification goals. Furthermore, their exclusion can significantly simplify the verification process. Consequently, the model shown in Listing 2.2 serves as the baseline for the verification efforts presented in this thesis.

CHAPTER 3

RELATED WORK

This section examines several studies relevant to this thesis.

Batz et al. [29] introduce a novel technique for verifying probabilistic programs by synthesising inductive invariants from source code. This technique is implemented in their tool *cegispro2*, which, like Caesar, aims to automate weakest preexpectation-based verification. A key difference, however, is that Caesar does not automate the derivation of loop invariants. Since determining loop invariants is a central challenge in this thesis, the approach by Batz et al. is particularly relevant. Their research includes a case study on the Bounded Retransmission Protocol (BRP), using a model nearly identical to the one in this thesis, excluding `sendPacket`. Their tool successfully synthesised an inductive wp-superinvariant for the configuration with $N \leq 8,000,000$ and $MAX \geq 5$, and they report the ability to also synthesise wp-subinvariants.

The remainder of this section examines several case studies on the Bounded Retransmission Protocol (BRP). Similar to this thesis, each of the related studies employs a distinct, simplified representation of BRP tailored to achieve specific research goals.

Helmink et al. [14] present one of the earliest studies on BRP, in which the protocol is modelled using Input/Output (I/O) automata theory. Their model is notably more detailed than the model used in this thesis, excluding only the disconnect feature and simplifying the timers of the original protocol. The correctness of this model is verified using the Coq proof development system [30]. The authors emphasise the potential of computer-assisted tools in streamlining and improving the efficiency of protocol verification tasks.

D'Argenio et al. [27] investigate the importance of timing in BRP to ensure reliability and efficiency. They model BRP as a network of timed automata, explicitly incorporating the sender, receiver, and communication channels. This model is verified using the model-checking tools SPIN [31] and UPPAAL [32]. Unlike the current study, which focuses on assessing the capabilities of a verification tool, their research primarily investigates BRP's verification itself, explaining their choice of a more complex BRP model.

In another study, D'Argenio et al. [16] introduce a novel technique to model check quantitative reachability properties on Markov decision processes. This approach relies on the iterative abstraction of the model to determine the appropriate level of detail required to verify a given property. Using this method, BRP is modelled as a probabilistic transition system and analysed against specific reachability conditions. The level of abstraction used in this study is consistent with the previously discussed works, including the sender,

receiver, and both communication channels in the model. While the focus on abstraction levels is relevant to this research, the primary objective remains the verification of BRP, which differs from the focus of this thesis.

Building on this research, a case study was conducted on the probabilistic model checker PRISM [33]. In this study, BRP is modelled as a Markov Decision Process, and reachability properties are verified using PRISM. The complexity of the BRP model is comparable to that used in the earlier study; however, the focus of this research now aligns with the goal of the current thesis—assessing the capabilities of a verification tool.

Spel [28] provides a formal framework to deduce the monotonicity of Markov Chains. In this research, BRP serves as a case study and is modelled as both a Markov Chain and the probabilistic program introduced in Section 2.4.1. In contrast to the previously discussed studies, this model abstracts away the internal mechanisms of BRP, focusing instead on the protocol's global properties, such as transmission success and the number of failed attempts. This model is closely related to the one used in the first section of the Stepwise Verification in this thesis (Section 5.1), although it was further abstracted during the verification process.

Batz et al. [34] developed a fully automated technique for verifying infinite-state programs, and used BRP as a case study. This study utilises a highly simplified pGCL model of the protocol, consisting of a single probabilistic choice within a while loop. Their approach enables the fully automated verification of an upper bound on the expected number of failed transmissions given an upper bound on the number of packets. Specifically, they verify that the expected number of failed transmissions is at most 1 if the number of packets to be sent is at most 3. The model used in this study is the most similar to the one employed in this thesis, although it is a simpler version. This alignment of research goals further strengthens the relevance of their work to this study.

Despite the extensive body of research on BRP, Caesar is a relatively new verification tool that has not yet been widely applied in verification studies. Its developers included several benchmarks in their work [15], three of which concern BRP. These benchmarks are based on the same simplified model used in the study by Batz et al. and verify an upper bound on the expected number of failed transmissions, given a maximum number of packets to be sent. This thesis aims to generalise these results by lifting the constraint on the number of packets, considering both lower and upper bounds on expectations, and verifying additional properties.

CHAPTER 4

METHODOLOGY

This chapter outlines the steps taken to address the research questions presented in Section 1 (page 3). The approach consists of the following key steps:

1. Abstraction of the Bounded Retransmission Protocol (BRP).
2. Manual weakest preexpectation calculations.
3. Translation from wp-calculus to Caesar procedures.
4. Verification using Caesar.

Each step builds upon its predecessors and is explained in more detail in the corresponding sections of this chapter. The relationship between these steps and the research questions is as follows: Step 1, the abstraction of BRP, supports the answer to sub-question 1. Step 2 (documented in Chapter 5) addresses sub-question 2, forms the foundation for Step 3, and contributes to sub-questions 3 and 4. Step 4, the practical verification using Caesar (documented in Chapter 6), supports the analysis of sub-questions 3, 4, and 5. The main Research Question can only be answered after all sub-questions have been addressed, and all research questions are discussed collectively in Chapter 7.

4.1 BRP Abstraction

The primary objective of this research is to evaluate the verification capabilities of Caesar rather than to verify BRP itself. Consequently, the initial model of BRP is based on the simplified representation shown in Figure 2.4 (page 18). This abstraction enables the verification of properties concerning the overall behaviour of BRP, such as the probability of success and the expected number of failed transmissions. The probabilistic program presented in Section 2.4.1 (page 18) represents one possible implementation of this simplified model of BRP in pGCL; however, it remains relatively complex, featuring a while loop with two if-statements and two probabilistic choices. Initial weakest preexpectation calculations were attempted on this model, but further simplification was needed to facilitate the verification process. The results of the calculations using this initial model are presented in Section 5.1.

The BRP model used in this thesis was derived by decomposing the initial program into two smaller subprograms, each containing a single while-loop, probabilistic choice and if-statement. The overarching program, `BRP`, models the entire protocol, invoking the `sendPacket` subprogram to model the transmission of individual packets. The resulting decomposed model is presented in Algorithm 1.

Algorithm 1 Pseudo code model of BRP

```
1:  procedure sendPacket
2:      failed ← 0
3:      success ← false
4:      while failed < MAX and not success do
5:          with probability p do success ← true else failed ← failed + 1
6:      end while
7:      return (success, failed)
8:  end procedure
9:  procedure BRP(N)
10:     sent ← 0
11:     success ← true
12:     totalFailed ← 0
13:     while sent < N and success do
14:         (success, failed) ← sendPacket
15:         totalFailed ← totalFailed + failed
16:         if success then
17:             sent ← sent + 1
18:         end if
19:     end while
20: end procedure
```

The sendPacket procedure models the receiver sending a single data packet until either an acknowledgement is received or the predefined limit on the maximum number of failed transmissions is reached. The variable `failed` represents the number of consecutive failed packet transmission attempts, which is upper-bounded by the variable `MAX`. The boolean variable `success` indicates whether the most recent single transmission attempt was successful. Finally, the variable `p` denotes the probability of a successful transmission, which includes both the probability of the packet being successfully transmitted from the sender to the receiver and the probability of receiving an acknowledgement in return.

The BRP procedure models the overall behaviour of the protocol when transmitting `N` packets. For each packet, it attempts transmission using sendPacket, terminating either when all packets have been successfully sent or when a packet transmission fails due to sendPacket exceeding the maximum number of consecutive failed transmissions. The variable `sent` tracks the number of successfully transmitted data packets, while `success` represents the success of the most recent transmission attempt. Additionally, `totalFailed` records the cumulative number of failed transmission attempts across all packets.

4.1.1 pGCL Implementation

The pGCL implementation of sendPacket, derived directly from the algorithmic description, is provided in Listing 4.1.

Listing 4.1: pGCL program `sendPacket`.

```
1 failed := 0;
2 success := false;
3 while (failed < MAX ∧ !success) {
4     success := true [p] failed := failed + 1;
5 }
```

In contrast, the BRP procedure in Algorithm 1 requires additional consideration when translated into pGCL, due to the language's lack of support for function calls. To address this, the result of the call to SENDPACKET in line 14 is modelled using two additional variables: s and f, which represent the probability of success and the expected number of failed transmissions, respectively, as returned by SENDPACKET. While these values are initially unspecified, this thesis aims to compute them as part of the verification process. The resulting pGCL implementation is presented in Listing 4.2.

LISTING 4.2: pGCL program BRP.

```
1 sent := 0;
2 success := true;
3 totalFailed := 0;
4
5 while (sent < N ∧ success) {
6     success := true [s] success := false;
7     totalFailed := totalFailed + f;
8
9     if (success) {
10         sent := sent + 1;
11    } else {}
12 }
```

Notably, both programs resemble the simple probabilistic program presented in Listing 2.1, commonly referred to as a geometric loop.

4.1.2 Verification Properties

Having established the pGCL model of BRP, relevant properties for verification can now be identified, thereby addressing RQ1.2. The theoretical verification of the following properties is documented in Chapter 5, and follows the methodology outlined in Section 4.2 of this chapter:

- Positive almost-sure termination:
  $\mathrm{ert}[\![\texttt{sendPacket}]\!](\mathbf{0}) \sqsubseteq \infty$ and $\mathrm{ert}[\![\texttt{BRP}]\!](\mathbf{0}) \sqsubseteq \infty$.
- The exact value of the probability of success:
  $\mathrm{wp}[\![\texttt{sendPacket}]\!]([\text{success}])$ and $\mathrm{wp}[\![\texttt{BRP}]\!]([\text{success}])$.
- The exact value of the expected number of failed transmissions:
  $\mathrm{wp}[\![\texttt{sendPacket}]\!](\text{failed})$ and $\mathrm{wp}[\![\texttt{BRP}]\!](\text{totalFailed})$.
- The exact value of the expected number of sent packets:
  $\mathrm{wp}[\![\texttt{BRP}]\!](\text{sent})$.

The omission of wlp-calculus in these verification goals is justified by the verification of positive almost-sure termination, which ensures the equivalence of wp- and wlp-calculus (see Theorem 3, page 11).

4.2 Manual Calculations

Before employing Caesar for verification, manual weakest preexpectation calculations were conducted to gain insight into the program and obtain a preexpectation to verify. Since `sendPacket` and `BRP` are similar programs, the steps for computing their weakest preexpectation given a postexpectation g are the same:

1. Work out $\text{wp}[\![\texttt{while-loop}]\!](g)$ to obtain the loop-characteristic function $\Phi_g(X)$.
2. Apply fixed-point iteration on $\Phi_g(X)$ to derive Φ_g^n.
3. Evaluate Φ_g^n for $n \to \infty$ to determine the supremum $\sup_{n\in\mathbb{N}} \Phi_g^n = \text{wp}[\![\texttt{while-loop}]\!](g)$.
4. Substitute this value into the equation for $\text{wp}[\![\texttt{program}]\!](g)$ to obtain the preexpectation.

For the pGCL program `sendPacket`, as presented in Listing 4.1, the equation $\text{wp}[\![\texttt{sendPacket}]\!](g)$ is formulated as follows:

$$\begin{aligned}
\text{wp}[\![\texttt{sendPacket}]\!](g) &= \text{wp}[\![\texttt{failed:=0;success:=false}]\!](\text{wp}[\![\texttt{while}_{\texttt{sp}}]\!](g)) \\
&= \text{wp}[\![\texttt{failed:=0;success:=false}]\!](\text{lfp } X.\Phi_{g,\texttt{sp}}(X)) \\
\Phi_{g,\texttt{sp}}(X) &= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X) \\
&\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot g \\
\text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X) &= p \cdot X[\text{success} := \mathit{true}] + (1-p) \cdot X[\text{failed} := \text{failed} + 1]
\end{aligned} \tag{4.1}$$

Analogously, the equation $\text{wp}[\![\texttt{BRP}]\!](g)$ for the pGCL program `BRP`, as defined in Listing 4.2, is given by:

$$\begin{aligned}
\text{wp}[\![\texttt{BRP}]\!](g) &= \text{wp}[\![\texttt{sent:=0;success:=true;totalFailed:=0}]\!](\text{wp}[\![\texttt{while}_{\texttt{brp}}]\!](g)) \\
&= \text{wp}[\![\texttt{sent:=0;success:=true;totalFailed:=0}]\!](\text{lfp } X.\Phi_{g,\texttt{brp}}(X)) \\
\Phi_{g,\texttt{brp}}(X) &= [\text{sent} < \text{N} \wedge \text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot g \\
\text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) &= s \cdot X[\text{sent} := \text{sent} + 1; \text{totalFailed} := \text{totalFailed} + f; \text{success} := \mathit{true}] \\
&\quad + (1-s) \cdot X[\text{totalFailed} := \text{totalFailed} + f; \text{success} := \mathit{false}]
\end{aligned} \tag{4.2}$$

Though these steps are quite straightforward, the least fixed point may be undecidable for complex programs. In such cases, super- or subinvariants may be used to establish upper or lower bounds on the loop preexpectation, as discussed in Section 2.2.2 (page 11).

4.3 From WP-Calculus to Caesar

When verifying programs that contain a while-loop in Caesar, it is necessary to apply an explicit proof rule. For the verification tasks considered in this thesis, the relevant proof rule is induction. This entails that, to verify an upper bound on the preexpectation, one must employ wp-calculus, a wp-superinvariant, and a Caesar `coproc`. Conversely, to verify a lower bound, wlp-calculus, a wlp-subinvariant, and a Caesar `proc` are required (see Table 2.4, page 16 for reference). Consequently, transitioning from wp-calculations to verification in Caesar entails two steps:

1. Identification of a suitable loop invariant.
2. Translation of the pGCL program to a HeyVL program.

4.3.1 Invariants

When reasoning about while-loops, Caesar requires an invariant that is maintained by each loop iteration [26]. This requirement is met in Listings 4.3 and 4.4 by preceding both while-loops with `@invariant(I)`. While substituting the calculated pre- and postexpectation into a HeyVL program is relatively straightforward, identifying an appropriate invariant, I, is more complex. The process of determining a suitable invariant consists of the following steps:

1. Propose a candidate invariant I.
2. Compute $\Phi(I)$.
3. Compare $\Phi(I)$ with I to determine whether it constitutes a superinvariant, subinvariant, both, or none (see Definition 10, page 12). If I does not satisfy the required properties, return to step 1 and select a new candidate.

The choice of whether a program requires a superinvariant or subinvariant depends on the proof rule: overapproximation requires a superinvariant, underapproximation requires a subinvariant, and exact verification requires both. While no universal procedure exists for selecting an appropriate candidate invariant, in practice, the equation for the supremum $\sup_{n\in\mathbb{N}}\Phi^n$ often serves as a useful starting point.

4.3.2 pGCL to HeyVL

Although the specific structure of the HeyVL program depends on the verification goal, the general form of the translated versions of `sendPacket` and `BRP` are presented in Listings 4.3 and 4.4. These programs incorporate pre- and postexpectations represented by the variables `preexp` and `postexp`, respectively, as well as an `@invariant` tag. Additionally, the placeholder `@tag` indicates the location of the appropriate calculus tag.

Listing 4.3: HeyVL program `sendPacket`.

```
domain Constants {
    func p(): UReal
    axiom pMax p() <= 1
    func MAX(): UInt
    axiom max_min MAX() > 0
}

@tag
proc sendPacket() -> (success: Bool, failed: UInt)
pre preexp
post postexp
{
    failed = 0
    success = false

    @invariant(invar)
    while (failed < MAX() && !success) {
        var cont: Bool = flip(p())
        if(cont) {
            success = true
        } else {
            failed = failed + 1
}}}
```

In the program described above, the probability of a successful transmission, `p`, and the maximum number of consecutive failed transmissions, `MAX`, are defined as constants using a domain block and axioms, rather than function parameters. For further details on these constructs, the reader is referred to the Caesar documentation [26]. This design choice is justified by the fact that, in practice, these values remain unchanged for each function call. The constant `p` is a real number bounded above by 1, as required for a probability, while `MAX` is an integer lower-bounded by 1, since a maximum failure count of 0 would render the protocol inoperative. Additionally, the probabilistic choice on line 4 of the pGCL program in Listing 4.1 is represented by lines 18-23 in the HeyVL translation.

Listing 4.4: HeyVL program `BRP`.

```
1  domain Constants {
2      func s(): UReal
3      axiom sMax s() <= 1
4
5      func f(): UInt
6  }
7
8  @tag
9  proc BRP(N: UInt) -> (totalFailed: UInt, success: Bool)
10 pre preexp
11 post postexp
12 {
13     var sent: UInt = 0
14     success = true
15     totalFailed = 0
16
17     @invariant(invar)
18     while(sent < N && success) {
19         success = flip(s())
20         totalFailed = totalFailed + f()
21
22         if (success) {
23             sent = sent + 1
24         } else {}
25     }
26 }
```

Analogous to the pGCL representation of BRP in Listing 4.2, the invocation of `sendPacket` is abstracted by lines 19 and 20. The real-type constant `s` represents the probability of success for `sendPacket`, and is upper-bounded by 1. Similarly, the integer-type constant `f` represents the expected value of `failed` in `sendPacket`. Unlike `p` and `MAX`, the number of packets to be sent, `N`, is defined as a function parameter, as this value may vary across different function calls.

If `sendPacket` were explicitly invoked rather than being abstracted using the constants `s` and `f`, lines 19-20 would be replaced by the code presented in Listing 4.5.

Listing 4.5: HeyVL call to function `sendPacket`.

```
1 var failed: UInt
2 success, failed = sendPacket()
3 totalFailed = totalFailed + failed
```

4.4 Caesar Verification

For this thesis, Caesar was initially installed via the Visual Studio Code extension described in option A of Caesar's installation guide [26]. Verification was performed by saving the file, which proved effective for cases where verification either succeeded or failed with a counterexample. However, for subsequent verification attempts, executing Caesar via the command-line interface became advantageous. Therefore, an additional installation was performed using the pre-built binary (option B.1 in the installation guide). Thereafter, Caesar was run from the command line with the probe flag enabled and a timeout of 20 seconds:

```
caesar verify <filename>.heyvl --timeout 20 --probe
```

The timeout was set to 20 seconds to increase the verification process's efficiency. This limit was based on observations that if verification does not complete within this time frame, it is unlikely to succeed. The probe flag enabled an output of the following form to the command line interface [26]:

```
Probe results for test.heyvl::test:
Has quantifiers: false
Detected theories: NIRA
 - complexity: Undecidable
 - rejected theories: LRA, LIA, LIRA, NRA, NIA
Number of arithmetic constants: 1
Number of Boolean constants: 4
Number of bit-vector constants: 0
Number of constants: 1
Number of expressions: 71
```

In general, verification in Caesar was conducted using a stepwise approach. The built-in error message and probe results provided guidance for subsequent verification steps. The stepwise verification is discussed in Chapter 6.

CHAPTER 5

THEORETICAL VERIFICATION

This chapter presents the theoretical verification approach introduced in the Methodology of this thesis, using the wp-calculus outlined in Section 2.2. The structure of this chapter is as follows:

1. **Initial:** An initial attempt to verify termination for the original BRP model.
2. **SendPacket:** Verification of the specified properties for the nested `sendPacket` program.
3. **BRP:** Verification of the same properties for the full `BRP` program, which includes `sendPacket`.
4. **Results:** A summary of the theoretical verification outcomes.
5. **Geometric Program:** Evaluation of expected values in a geometric loop to support and validate the manual calculations.

The first subsection addresses the initial verification attempt for termination, based on the initial BRP model shown in Listing 5.1. To establish termination, it suffices to prove almost-sure termination or positive almost-sure termination (see Definition 12, page 14 for reference). Almost-sure termination is the weaker of the two classifications and was initially expected to be easier to verify. However, these verification attempts were unsuccessful, as computing the supremum for the loop-characteristic function proved infeasible. Caesar's loop-unrolling feature was used to identify potential errors in the manual computations, but none were found, necessitating an alternative approach.

Consequently, the model was decomposed into two separate programs: `sendPacket` and `BRP`, where the latter invokes the former. These abstractions, introduced in Section 4.1, allow each verification task to be performed independently on the two programs.

The verification focuses on the following properties:

- Termination.
- Probability of success.
- Expected number of failed transmissions.
- Expected number of sent packets.

Termination is revisited for the simplified model in subsections 5.2.1 and 5.3.1. Rather than proving almost-sure termination, as was attempted in the first section of this chapter, the revisited approach establishes positive almost-sure termination using a modified version

of ert-calculus (see Table 2.3 for reference). In this modified calculus, a program's runtime is incremented solely for while-loop iterations, effectively counting loop executions rather than the overall program runtime. The abstracted model and modified ert-calculus significantly simplified the calculus compared to the initial attempt, enabling the verification of positive almost-sure termination.

The remaining subsections in Sections 5.2 and 5.3 address the verification of the remaining properties: the probability of success, the expected number of failed transmissions, and the expected number of sent packets. The computed preexpectations and suprema are summarised in Subsection 5.4.

Finally, when attempting to verify expected number of failed transmissions and sent packets using Caesar, the computed preexpectation could not be verified as an upper or lower bound. To validate these manually computed values, the final section of this chapter, Section 5.5, evaluates the expected number of trials and failures for a simple geometric loop. These results are then related to `sendPacket` and `BRP`, both of which resemble a geometric loop, thereby providing an additional layer of verification for the manual calculations.

5.1 Initial Attempt

Initial weakest preexpectation calculations were conducted on a relatively complex model, represented by the pGCL program shown in Listing 5.1.

Listing 5.1: An initial pGCL model of `BRP`.

```
failed := 0;
sent := 0;
totalFailed := 0;

while (sent < N ∧ failed < MAX) {
    lost_frame := 0 [p] lost_frame := 1;
    lost_ack := 0 [p] lost_ack := 1;
    if (lost_frame ∨ lost_ack) {
        failed := failed + 1;
    } else {
        sent := sent + 1;
        totalFailed := totalFailed + failed
        failed := 0;
    }
}
totalFailed := totalFailed + failed
```

In this program, `N` denotes the total number of packets to be transmitted, while the variable `failed` tracks the number of consecutive failed transmission attempts, constrained by the upper bound `MAX`. The variable `sent` represents the number of successfully transmitted packets, and `totalFailed` records the cumulative number of failed transmissions. The probability of successful packet transmission over a communication channel is represented by the variable `p`, with the boolean variable `lost_frame` indicating whether a data frame was lost during transmission, and `lost_ack` doing so for acknowledgement packets.

To establish the termination of the model P, it suffices to demonstrate that either $\mathrm{wp}[\![P]\!](\mathbf{1}) = \mathbf{1}$ or $\mathrm{ert}[\![P]\!](\mathbf{0}) \sqsubset \infty$ (see Definition 12, page 14). The former approach (almost-sure termination) was opted for, as it is the weaker of the two classifications. The steps listed in Section 4.2 (page 25) were followed, yielding the following equations for the

loop-characteristic function:

$$\Phi(X) = [\text{sent} < \text{N} \wedge \text{failed} < \text{MAX}] \cdot \text{wp}[\![\texttt{body}]\!](X) + [\text{sent} \geq \text{N} \vee \text{failed} \geq \text{MAX}]$$

$$\text{wp}[\![\text{body}]\!](X) = p^2 \cdot X[\text{sent} := \text{sent} + 1; \text{failed} := 0] + (1+p) \cdot (1-p) \cdot X[\text{failed} := 0]$$

Using these equations, fixed-point iteration was carried out to compute Φ^1 through Φ^6. The results for the first five iterations are provided in Appendix A.1. Given that the equation Φ^6 spans more than a page, it has been omitted from this thesis.

Following an analysis of the equations for Φ^1 through Φ^3, an equation for Φ^n was derived. However, inconsistencies were observed in Φ^4, which introduced terms that did not align with the established pattern. When Φ^n was updated and compared with Φ^5, further inconsistencies emerged. A subsequent refinement of Φ^n and comparison against Φ^6 again resulted in the introduction of new terms. Two potential explanations exist for these discrepancies: (1) the equations for $\Phi^1 - \Phi^6$ contain errors, (2) Φ^n is too complex to derive using this method. The first possibility was within our control; thus, Caesar was utilised to verify Φ^1 through Φ^5.

5.1.1 Loop-unrolling using Caesar

Caesar's loop-unrolling feature can be used to refute a candidate upper bound for a while-loop. This feature enables the replacement of a while-loop with a fixed number of iterations k. If the candidate upper bound is shown to be invalid for this specific number of loop iterations, it is guaranteed not to hold for the preexpectation of the while-loop (see Theorem 6). However, if verification succeeds, it does not imply that the given preexpectation holds for the while-loop.

Theorem 6 (Bounded Model Checking for pGCL). [34]
For probabilistic program $C_{\text{loop}} = \texttt{while}(\varphi)\{C\}$ and loop-characteristic function Φ:

$$\exists n \in \mathbb{N} : \Phi^n(0) \not\sqsubseteq f \iff \text{wp}[\![C_{\text{loop}}]\!](g) \not\sqsubseteq f.$$

In this thesis, loop unrolling is employed to under-approximate the least fixed-point. For this purpose, a `coproc` is used, and the *terminator* is set to 0. The HeyVL code used in this analysis is provided in Listing 5.2 and is available for download from the accompanying repository [35].

Listing 5.2: Loop-unrolling in Caesar.

```
domain Constants {
    func p(): UReal
    axiom pMax p() <= 1

    func MAX(): UInt
    axiom MAXmin MAX() > 0

    func N(): UInt
    axiom Nmin N() > 0
}

coproc unroll(ghost_sent: UInt, ghost_failed: UInt,
    ghost_totalFailed: UInt) -> (sent: UInt, totalFailed: UInt,
    failed: UInt)
```

```
13 pre Φ^n
14 post [ghost_sent == sent && ghost_failed == failed]
15 {
16 failed = 0
17 totalFailed = 0
18 sent = 0
19
20 @unroll(k,0)
21 while (sent < N()) && (failed < MAX()) {
22     var q: UReal = 1-p()
23     var lost_frame: Bool = flip(q)
24     var lost_ack: Bool = flip(q)
25     if lost_frame || lost_ack {
26         failed = failed + 1
27     } else {
28         totalFailed = totalFailed + 1
29         failed = 0
30         sent = sent + 1
31     }
32 }
```

In this program, ghost variables allow the values of variables initialised in the output list to be referenced in the preexpectation. These ghost variables are equated to their corresponding variable in the postexpectation, as illustrated in line 14 of Listing 5.2.

To assess the correctness of the equations for Φ^n, the variable k in the program was assigned the corresponding value of n in Φ^n, and the preexpectation was replaced by the relevant equation. Listing 5.3 presents the HeyVL translation of Φ^2; subsequent iterations were similarly translated.

Listing 5.3: Φ^2 in HeyVL.

```
1 p()*p()*[ghost_sent+1 == N() && ghost_failed < MAX()] + (1-p())*(1+
    p())*[ghost_sent < N() && ghost_failed + 1 == MAX()] + [
    ghost_sent >= N() || ghost_failed >= MAX()]
```

Loop-unrolling verified the absence of errors in the equations for Φ^1 through Φ^4. Notably, when all four procedures were included in a single HeyVL file, verification was unsuccessful, terminating with the error message "Verification had an error: interrupted". However, verification succeeded when the procedures were tested separately.

Although the absence of errors in the computed iterations of Φ may initially appear encouraging, it does not facilitate the computation of the least fixed point. While further loop unrolling is theoretically possible, computing the least fixed point may require an infinite number of iterations, rendering the problem undecidable. As a result, an alternative approach was adopted: the simplification of the BRP model. The simplified model isolates the process of sending a single packet, `sendPacket`, which is invoked by `BRP`. By decomposing the program in this manner, two simple programs are created, both of which resemble a geometric loop. This abstraction of BRP was explained in further detail in Section 4.1 (page 22).

5.2 SendPacket

This section, along with subsequent sections, utilises the simplified BRP model provided in Listings 4.1 and 4.2 on page 23.

This section contains the theoretical verification of termination, the probability of success, and the expected number of failed transmissions for `sendPacket`. The expected number of sent packets is notably absent from this list, as this is a property exclusively of `BRP`.

Each subsection approaches a the verification of a property by following the steps outlined in the methodology of this thesis (Chapter 4.2, page 25):

1. Work out $\text{wp}[\![\texttt{while}_{\texttt{sp}}]\!](g)$ to obtain the loop-characteristic function $\Phi_{g,\texttt{sp}}(X)$.
2. Apply fixed-point iteration on $\Phi_{g,\texttt{sp}}(X)$ to derive $\Phi^n_{g,\texttt{sp}}$.
3. Evaluate $\Phi^n_{g,\texttt{sp}}$ for $n \to \infty$ to determine the supremum $\sup_{n\in\mathbb{N}}\Phi^n_{g,\texttt{sp}} = \text{wp}[\![\texttt{while}_{\texttt{sp}}]\!](g)$.
4. Substitute this value into the equation for $\text{wp}[\![\texttt{program}]\!](g)$ to obtain the preexpectation.

Throughout this section, the notation $\Phi_{g,\texttt{sp}}$ is abbreviated to Φ to improve readability. The first subsection presents each verification step in detail. For the subsequent sections, the corresponding fixed-point iterations are provided in Appendix A to avoid redundancy. Additionally, extensive supremum simplifications are included in Appendix B.

5.2.1 Termination

The objective of this section is to verify positive almost-sure termination (PAST), expressed as $\text{ert}[\![\texttt{sendPacket}]\!](\mathbf{0}) \sqsubset \infty$. To determine the number of while-loop iterations, the ert-calculus outlined in Table 2.3 (page 14) is modified such that 1 is added to the expected runtime for $C = \text{while } (\varphi)\{P\}$, but nowhere else. Calculations for $\text{ert}[\![\texttt{sendPacket}]\!](\mathbf{0})$ using this modified calculus yielded the following equation for the loop-characteristic function:

Loop-characteristic function

$$\Phi(X) = \mathbf{1} + [\text{failed} < \text{MAX} \land \neg\text{success}] \cdot \text{ert}[\![\texttt{body}_{\texttt{sp}}]\!](X)$$

Where the equation for $\text{ert}[\![\texttt{body}_{\texttt{sp}}]\!](X)$ is identical to $\text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X)$ in Equation 4.1 (pg 25).

Fixed-point iteration

Fixed-point iteration was subsequently applied to derive the equations for Φ^1 through Φ^4. For better readability, the notation $\Phi^n(\mathbf{0})$ is simplified to Φ^n:

$$\begin{aligned}
\Phi^1 &= \mathbf{1} \\
\Phi^2 &= \mathbf{1} + [\text{failed} < \text{MAX} \land \neg\text{success}] \\
\Phi^3 &= \mathbf{1} + [\text{failed} < \text{MAX} \land \neg\text{success}] \\
&\quad + (1-p) \cdot [\text{failed} + 1 < \text{MAX} \land \neg\text{success}] \\
\Phi^4 &= \mathbf{1} + [\text{failed} < \text{MAX} \land \neg\text{success}] \\
&\quad + (1-p) \cdot [\text{failed} + 1 < \text{MAX} \land \neg\text{success}] \\
&\quad + (1-p)^2 \cdot [\text{failed} + 2 < \text{MAX} \land \neg\text{success}]
\end{aligned}$$

Based on these equations, the equation for Φ^n is formulated as

$$\Phi^n = \mathbf{1} + [\neg\text{success}] \cdot \sum_{i=0}^{n-2} (1-p)^i \cdot [\text{failed} + i < \text{MAX}].$$

Upper bound on the preexpectation

I then evaluated Φ^n for $n \to \infty$ to compute the supremum $\sup_{n\in\mathbb{N}}\Phi^n$. Since the goal is to verify positive almost-sure termination, rather than computing the exact supremum, an upper bound suffices:

$$\begin{aligned}
\sup_{n\in\mathbb{N}}\Phi^n &= \mathbf{1} + [\neg\text{success}] \cdot \sum_{i=0}^{\infty}(1-p)^i \cdot [\text{failed} + i < \text{MAX}] \\
&\sqsubseteq \mathbf{1} + [\neg\text{success}] \cdot \sum_{i=0}^{\infty}(1-p)^i \\
&= \mathbf{1} + [\neg\text{success}] \cdot \frac{1}{1-(1-p)} \\
&= \mathbf{1} + [\neg\text{success}] \cdot \frac{1}{p}
\end{aligned}$$

Remark 1 . *This simplification relies on the assumption that $p > 0$.*

This assumption is justified, as a zero probability of successful transmission would render the application of BRP unnecessary and impractical in any realistic scenario. To confirm that this is a valid upper bound, $\Phi(I)$ is computed for candidate wp-superinvariant I:

$$\begin{aligned}
I &= \mathbf{1} + [\neg\text{success}] \cdot \frac{1}{p} \\
\Phi(I) &= \mathbf{1} + [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \frac{1}{p}
\end{aligned}$$

Since $\Phi(I) \sqsubseteq I$, it follows that I is a wp-superinvariant and, consequently, an upper bound on the preexpectation of the while-loop in `sendPacket` (c.f. Definition 10 and Theorem 4, for reference). Thus, the upper bound for $\text{ert}[\![\texttt{sendPacket}]\!](\mathbf{0})$ is computed as follows:

$$\begin{aligned}
\text{ert}[\![\texttt{sendPacket}]\!](\mathbf{0}) &= \text{ert}[\![\texttt{failed:=0;success:=false}]\!](\text{ert}[\![\texttt{while}_{\texttt{sp}}]\!](\mathbf{0})) \\
&\sqsubseteq \text{ert}[\![\texttt{failed:=0;success:=false}]\!](\mathbf{1} + [\neg\text{success}] \cdot \frac{1}{p}) \\
&= \mathbf{1} + \frac{1}{p} \\
&\sqsubset \infty.
\end{aligned}$$

5.2.2 Probability of Success

The calculations in this and subsequent sections utilise the wp-calculus framework outlined in Table 2.1 on page 7. The aim of these computations is to derive the value of $\text{wp}[\![\texttt{sendPacket}]\!]([\text{success}])$.

Loop-characteristic function

$$\begin{aligned}
\Phi(X) &= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X) + [\text{success}] \\
\text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X) &= p \cdot X[\text{success} := \mathit{true}] + (1-p) \cdot X[\text{failed} := \text{failed} + 1].
\end{aligned}$$

The equation for $\text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X)$ was given in Equation 4.1 on page 25, and repeated here as a reminder. In subsequent sections, it will not be written out, as it remains the same.

Fixed-point iteration

Using these equations, fixed-point iteration was carried out to compute Φ^1 through Φ^4 (see Appendix A.2.1). Based on these equations, the expression for Φ^n is formulated as:

$$\Phi^n = [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \sum_{i=0}^{n-2} p \cdot (1-p)^i \cdot [\text{failed} + i < \text{MAX}] + [\text{success}].$$

Determine the supremum

The equation for Φ^n is evaluated for $n \to \infty$ to compute the supremum $\sup_{n\in\mathbb{N}}\Phi^n$.

$$\begin{aligned}
&\sup_{n\in\mathbb{N}}\Phi^n \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \sum_{i=0}^{\infty} p \cdot (1-p)^i \cdot [\text{failed} + i < \text{MAX}] + [\text{success}] \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \sum_{i=0}^{\text{MAX}-\text{failed}-1} p \cdot (1-p)^i + [\text{success}]. \qquad \text{(Restrict the sum)}
\end{aligned}$$

To restrict the sum, it must be ensured that the upper bound remains greater than or equal to the lower bound, i.e. $\text{MAX} - \text{failed} - 1 \geq 0$. This condition is satisfied by analysing the guard preceding the summation:

$$\begin{aligned}
\text{sent} < \text{N} &\iff 0 < \text{N} - \text{sent} \\
&\iff 0 \leq \text{N} - \text{sent} - 1.
\end{aligned}$$

As shown, this inequality holds under the given guard in the equation for S_1, justifying the restriction of the summation.

The resulting summation resembles the well-established geometric sum formula presented in Equation 7.4.

$$S_n = \sum_{k=0}^{n} a \cdot r^k = \begin{cases} a \cdot (n+1) & r = 1, \\ a \cdot \frac{1-r^{n+1}}{1-r} & |r| < 1. \end{cases} \tag{5.1}$$

In this case, however, the coefficient a is defined as $(1-r) \cdot b$, where b is an arbitrary term. Substituting this into Equation 7.4 allows the case distinction to be eliminated, yielding a simplified and unified expression:

$$\sum_{k=0}^{n} (1-r) \cdot b \cdot r^k = b \cdot (1 - r^{n+1}). \tag{5.2}$$

To justify this simplification, we demonstrate that Equation 5.2 is equivalent to Equation 7.4, under the substitution $a = (1-r) \cdot b$, in both cases:

Case 1: $r = 1$

$$\begin{aligned}
\sum_{k=0}^{n} a \cdot r^k &= a \cdot (n+1) && \text{(Equation 7.4)} \\
&= (1-r) \cdot b \cdot (n+1) && \text{(Substitute } a\text{)} \\
&= 0 \cdot b \cdot (n+1) && \text{(Apply } r = 1\text{)} \\
&= 0 && \text{(Simplify)}
\end{aligned}$$

$$\begin{aligned}\sum_{k=0}^{n}(1-r)\cdot b\cdot r^k &= b\cdot(1-r^{n+1}) && \text{(Equation 5.2)}\\ &= b\cdot(1-1^{n+1}) && \text{(Apply } r=1\text{)}\\ &= 0 && \text{(Simplify)}\end{aligned}$$

Case 2: $|r|<1$

$$\begin{aligned}\sum_{k=0}^{n} a\cdot r^k &= a\cdot\frac{1-r^{n+1}}{1-r} && \text{(Equation 7.4)}\\ &= (1-r)\cdot b\cdot\frac{1-r^{n+1}}{1-r} && \text{(Substitute } a\text{)}\\ &= b\cdot(1-r^{n+1}) && \text{(Simplify)}\\ &= \sum_{k=0}^{n}(1-r)\cdot b\cdot r^k && \text{(Equation 5.2)}\end{aligned}$$

Having demonstrated the equivalence of the two formulas, we are justified in applying Equation 5.2 in our simplification.

$$\begin{aligned}\sup_{n\in\mathbb{N}}\Phi^n &= [\text{failed} < \text{MAX} \wedge \neg\text{success}]\cdot\sum_{i=0}^{\text{MAX}-\text{failed}-1} p\cdot(1-p)^i + [\text{success}]\\ &= [\text{failed} < \text{MAX} \wedge \neg\text{success}]\cdot(1-(1-p)^{\text{MAX}-\text{failed}}) + [\text{success}]\end{aligned}$$

Solve for the preexpectation

$$\begin{aligned}\text{wp}[\![\texttt{sendPacket}]\!]([\text{success}]) &= \text{wp}[\![\texttt{failed:=0;success:=false}]\!](\text{wp}[\![\texttt{while}_{\texttt{sp}}]\!]([\text{success}]))\\ &= \text{wp}[\![\texttt{failed:=0;success:=false}]\!](\sup_{n\in\mathbb{N}}\Phi^n)\\ &= 1-(1-p)^{\text{MAX}}\end{aligned}$$

5.2.3 Expected Number of Failures

The aim of these computations is to derive the value of $\text{wp}[\![\texttt{sendPacket}]\!](\text{failed})$.

Loop-characteristic function and fixed-point iteration

$$\Phi(X) = [\text{failed} < \text{MAX} \wedge \neg\text{success}]\cdot \text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X) + [\text{failed} \geq \text{MAX} \vee \text{success}]\cdot\text{failed}$$

The equation $\text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X)$ is provided in Equation 4.1 on page 25.

Fixed-point iteration

Using these equations, fixed-point iteration was carried out to compute Φ^1 through Φ^4 (see Appendix A.2.2). Based on these equations, the expression for Φ^n is formulated as:

$$\begin{aligned}\Phi^n = [\text{failed} < \text{MAX} \wedge \neg\text{success}]\cdot\Bigg(&\sum_{i=0}^{n-2}[\text{failed}+i < \text{MAX}]\cdot p\cdot(1-p)^i\cdot(\text{failed}+i)\\ &+\sum_{i=0}^{n-2}[\text{failed}+i+1 = \text{MAX}]\cdot(1-p)^{i+1}\cdot(\text{failed}+i+1)\Bigg)\\ &+[\text{failed} \geq \text{MAX} \vee \text{success}]\cdot\text{failed}\end{aligned}$$

Determine the supremum

The equation for Φ^n is evaluated for $n \to \infty$ to compute the supremum $\sup_{n\in\mathbb{N}}\Phi^n$. The simplification of $\sup_{n\in\mathbb{N}}\Phi^n$ can be seen in Appendix B.1, the result of which is the following:

$$\begin{aligned}\sup_{n\in\mathbb{N}}\Phi^n = {} & [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \left(\text{failed} + \frac{(1-p) - (1-p)^{\text{MAX}-\text{failed}+1}}{p}\right) \\ & + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed}\end{aligned}$$

Solve for the preexpectation

$$\begin{aligned}\text{wp}[\![\texttt{sendPacket}]\!](\text{failed}) &= \text{wp}[\![\texttt{failed:=0;success:=false}]\!](\text{wp}[\![\texttt{while}_{\texttt{sp}}]\!](\text{failed})) \\ &= \text{wp}[\![\texttt{failed:=0;success:=false}]\!](\sup_{n\in\mathbb{N}}\Phi^n) \\ &= \frac{(1-p) - (1-p)^{\text{MAX}+1}}{p}\end{aligned}$$

5.3 BRP

Having completed the verification of each property for `sendPacket`, we now proceed to verify the corresponding properties for `BRP`. The same verification procedure as outlined in the previous sections is followed. Throughout this section, the notation Φ is used as shorthand for $\Phi_{g,\texttt{brp}}$ to improve readability. As before, only the fixed-point iteration and supremum simplification for the first property are presented here to avoid redundancy.

5.3.1 Termination

The verification process outlined in the previous section was applied to establish an upper bound on the expected number of loop iterations of `BRP`.

Loop-characteristic function

$$\Phi(X) = \mathbf{1} + [\text{sent} < N \wedge \text{success}] \cdot \text{ert}[\![\texttt{body}_{\texttt{brp}}]\!](X)$$

Where the equation for $\text{ert}[\![\texttt{body}_{\texttt{brp}}]\!](X)$ is identical to $\text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X)$ in Equation 4.2 (pg 25).

Fixed-point iteration

For readability, the notation $\Phi^n(\mathbf{0})$ is abbreviated to Φ^n.

$$\begin{aligned}\Phi^1 &= \mathbf{1} \\ \Phi^2 &= \mathbf{1} + [\text{sent} < N \wedge \text{success}] \\ \Phi^3 &= \mathbf{1} + [\text{sent} < N \wedge \text{success}] + q \cdot [\text{sent} + 1 < N \wedge \text{success}] \\ \Phi^4 &= \mathbf{1} + [\text{sent} < N \wedge \text{success}] + q \cdot [\text{sent} + 1 < N \wedge \text{success}] \\ &\quad + q^2 \cdot [\text{sent} + 2 < N \wedge \text{success}]\end{aligned}$$

Based on the equations for Φ^1 through Φ^4, the equation for Φ^n is formulated as follows.

$$\Phi^n = \mathbf{1} + [\text{success}] \cdot \sum_{i=0}^{n-2} q^i \cdot [\text{sent} + i < N]$$

Upper bound on the preexpectation

The equation for Φ^n is evaluated for $n \to \infty$ to compute an upper bound to the supremum:

$$\begin{aligned}\sup_{n\in\mathbb{N}}\Phi^n &= \mathbf{1} + [\text{success}] \cdot \sum_{i=0}^{\infty} s^i \cdot [\text{sent} + i < N] \\ &\sqsubseteq \mathbf{1} + [\text{success}] \cdot \sum_{i=0}^{\infty} s^i \\ &= \mathbf{1} + [\text{success}] \cdot \frac{1}{1-s}\end{aligned}$$

Remark 2 . *This simplification relies on the assumption that $s < 1$.*

For `sendPacket` to yield a preexpectation of 1 given the postexpectation [success], the probability of a successful packet transmission must be 1. Analogously to a zero probability of successful packet transmission, it is unlikely that BRP would be employed in such a scenario.

To confirm that the computed upper bound to the supremum is a valid upper bound of the while-loop, $\Phi(I)$ is computed for candidate invariant I:

$$\begin{aligned}I &= \mathbf{1} + [\text{success}] \cdot \frac{1}{1-s} \\ \Phi(I) &= \mathbf{1} + [\text{sent} < N \wedge \text{success}] \cdot \frac{s}{1-s}\end{aligned}$$

From these equations, it follows that $\Phi(I) \sqsubseteq I$, thereby verifying that I is a superinvariant and thus serves as an upper bound for the preexpectation of the while-loop in `BRP` (c.f. Definition 10 and Theorem 4, for reference). Finally, the upper bound for $\text{ert}[\![\texttt{BRP}]\!](\mathbf{0})$ is determined as follows:

$$\begin{aligned}\text{ert}[\![\texttt{BRP}]\!](\mathbf{0}) &= \text{ert}[\![\texttt{sent:=0;success:=true;totalFailed:=0}]\!](\text{ert}[\![\texttt{while}_{\texttt{brp}}]\!](\mathbf{0})) \\ &\sqsubseteq \text{ert}[\![\texttt{sent:=0;success:=true}]\!](\mathbf{1} + [\text{success}] \cdot \frac{1}{1-s}) \\ &= \mathbf{1} + \frac{1}{1-s} \\ &\sqsubset \infty\end{aligned}$$

5.3.2 Probability of Success

The aim of this section is to compute the value of $\text{wp}[\![\texttt{BRP}]\!]([\text{success}])$.

Loop-characteristic function

$$\Phi(X) = [\text{sent} < \text{N} \wedge \text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) + [\text{sent} \geq \text{N} \wedge \text{success}]$$

The equation for $\text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X)$ was given in Equation 4.2 on page 25, and repeated here as a reminder. In subsequent sections, it will not be written out, as it remains the same.

Fixed-point iteration

Using this equation, I performed fixed-point iteration to determine Φ^1 through Φ^4 (see Appendix A.3.1) Based on these equations, the expression for Φ^n is formulated as:

$$\Phi^n = [\text{sent} < \text{N} \wedge \text{success}] \cdot \sum_{i=0}^{n-1} s^i \cdot [\text{sent} + i = \text{N}] + [\text{sent} \geq \text{N} \wedge \text{success}]$$

Determine the supremum

The equation for Φ^n is evaluated for $n \to \infty$ to compute the supremum $\sup_{n\in\mathbb{N}}\Phi^n$:

$$\begin{aligned}\sup_{n\in\mathbb{N}}\Phi^n &= [\text{sent} < \text{N} \wedge \text{success}] \cdot \sum_{i=0}^{\infty} s^i \cdot [\text{sent} + i = \text{N}] + [\text{sent} \geq \text{N} \wedge \text{success}] \\ &= [\text{sent} < \text{N} \wedge \text{success}] \cdot s^{\text{N}-\text{sent}} + [\text{sent} \geq \text{N} \wedge \text{success}]\end{aligned}$$

Solve for the preexpectation

$$\begin{aligned}\text{wp}[\![\texttt{BRP}]\!]([\text{success}]) &= \text{wp}[\![\texttt{sent:=0;success:=true;totalFailed:=0}]\!](\\ &\qquad \text{wp}[\![\texttt{while}_{\texttt{brp}}]\!]([\text{success}])) \\ &= \text{wp}[\![\texttt{sent:=0;success:=true;totalFailed:=0}]\!](\sup_{n\in\mathbb{N}}\Phi^n) \\ &= s^{\text{N}}\end{aligned}$$

Where $s = \text{wp}[\![\texttt{sendPacket}]\!]([\text{success}]) = 1 - (1-p)^{\text{MAX}}$

5.3.3 Expected Number of Failures

This section aims to obtain the value of $\text{wp}[\![\texttt{BRP}]\!](\text{totalFailed})$.

Loop-characteristic function

$$\Phi(X) = [\text{sent} < \text{N} \wedge \text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed}$$

The equation for $\text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X)$ is provided in Equation 4.2 on page 25.

Fixed-point iteration

Using these equations, fixed-point iteration was carried out to compute Φ^1 through Φ^4 (see Appendix A.3.2). Based on these equations, the expression for Φ^n is formulated as:

$$\begin{aligned}\Phi^n = [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(&\sum_{i=0}^{n-2} [\text{sent} + i < \text{N}] \cdot s^i \cdot (1-s) \cdot (\text{totalFailed} + (i+1) \cdot f) \\ &+ \sum_{i=0}^{n-2} [\text{sent} + i + 1 = \text{N}] \cdot s^{i+1} \cdot (\text{totalFailed} + (i+1) \cdot f) \Bigg) \\ &+ [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{totalFailed}\end{aligned}$$

Determine the supremum

The equation for Φ^n is evaluated for $n \to \infty$ to compute the supremum $\sup_{n\in\mathbb{N}}\Phi^n$. The simplification of $\sup_{n\in\mathbb{N}}\Phi^n$ can be seen in Appendix B.2, the result of which is the following:

$$\begin{aligned}\sup_{n\in\mathbb{N}}\Phi^n = [\text{success} \wedge \text{sent} < \text{N}] \cdot &\left(\text{totalFailed} + f \cdot \frac{1 - s^{\text{N}-\text{sent}}}{1-s}\right) \\ &+ [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{totalFailed}\end{aligned}$$

Solve for the preexpectation

$$\begin{aligned}\mathrm{wp}[\![\texttt{BRP}]\!](\text{totalFailed}) &= \mathrm{wp}[\![\texttt{sent:=0;success:=true;totalFailed:=0}]\!]\big(\\ &\qquad \mathrm{wp}[\![\texttt{while}_{\texttt{brp}}]\!](\text{totalFailed})\big)\\ &= \mathrm{wp}[\![\texttt{sent:=0;success:=true;totalFailed:=0}]\!](\sup_{n\in\mathbb{N}}\Phi^n)\\ &= f\cdot\frac{1-s^{\mathrm{N}}}{1-s}\end{aligned}$$

Where $s = \mathrm{wp}[\![\texttt{sendPacket}]\!]([\text{success}]) = 1-(1-p)^{\mathrm{MAX}}$

$$f = \mathrm{wp}[\![\texttt{sendPacket}]\!](\text{failed}) = \frac{q-q^{\mathrm{MAX}+1}}{p}$$

5.3.4 Expected Number of Sent Packets

The aim of these computations is therefore to derive the value of $\mathrm{wp}[\![\texttt{BRP}]\!](\text{sent})$.

Loop-characteristic function

$$\Phi(X) = [\text{sent} < \mathrm{N} \wedge \text{success}]\cdot \mathrm{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) + [\text{sent} \geq \mathrm{N} \vee \neg\text{success}]\cdot\text{sent}$$

The equation for $\mathrm{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X)$ is provided in Equation 4.2 on page 25.

Fixed-point iteration

Using these equations, fixed-point iteration was carried out to compute Φ^1 through Φ^4 (see Appendix A.3.3). Based on these equations, the expression for Φ^n is formulated as:

$$\begin{aligned}\Phi^n = [\text{success}\wedge\text{sent}<\mathrm{N}]\cdot\Bigg(&\sum_{i=0}^{n-2}[\text{sent}+i<\mathrm{N}]\cdot s^i\cdot(1-s)\cdot(\text{sent}+i)\\ &+\sum_{i=0}^{n-2}[\text{sent}+i+1=\mathrm{N}]\cdot s^{i+1}\cdot(\text{sent}+i+1)\Bigg)\\ &+[\neg\text{success}\vee\text{sent}\geq\mathrm{N}]\cdot\text{sent}\end{aligned}$$

Determine the supremum

The equation for Φ^n is evaluated for $n\to\infty$ to compute the supremum $\sup_{n\in\mathbb{N}}\Phi^n$. The simplification of $\sup_{n\in\mathbb{N}}\Phi^n$ can be seen in Appendix B.3, the result of which is the following equation:

$$\begin{aligned}\sup_{n\in\mathbb{N}}\Phi^n = [\text{success}\wedge\text{sent}<\mathrm{N}]\cdot&\left(\text{sent}+\frac{s(1-s^{\mathrm{N}-\text{sent}})}{1-s}\right)\\ &+[\neg\text{success}\vee\text{sent}\geq\mathrm{N}]\cdot\text{sent}\end{aligned}$$

Solve for the preexpectation

$$\begin{aligned}\mathrm{wp}[\![\texttt{BRP}]\!](\text{sent}) &= \mathrm{wp}[\![\texttt{sent:=0;success:=true;totalFailed:=0}]\!](\mathrm{wp}[\![\texttt{while}_{\texttt{brp}}]\!](\text{sent}))\\ &= \mathrm{wp}[\![\texttt{sent:=0;success:=true}]\!](\sup_{n\in\mathbb{N}}\Phi^n)\\ &= \frac{s\cdot(1-s^{\mathrm{N}})}{1-s}\end{aligned}$$

Where $s = \mathrm{wp}[\![\texttt{sendPacket}]\!]([\text{success}]) = 1-(1-p)^{\mathrm{MAX}}$

5.4 Results

Termination, the probability of success, and the expected number of failed and sent packets have all been successfully verified through weakest preexpectation calculations using a pGCL model of BRP. A summary of the theoretical verification results from the preceding sections is presented in Tables 5.1 and 5.2. The tables list both the computed preexpectations and the suprema corresponding to their while-loops, as these are necessary for translation to Caesar-based verification.

calc	g	**supremum** calc$[\![\texttt{while}_{\texttt{sp}}]\!](g)$	**preexpectation** calc$[\![\texttt{sendPacket}]\!](g)$
ert	$\mathbf{0}$	$\sqsubseteq \mathbf{1} + [\neg\text{success}] \cdot \frac{1}{p}*$	$\sqsubseteq \mathbf{1} + \frac{1}{p}*$
wp	[success]	$= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \Big($ $1 - (1-p)^{\text{MAX}-\text{failed}}\Big) + [\text{success}]$	$= 1 - (1-p)^{\text{MAX}}$
wp	failed	$= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \Big($ $\text{failed} + \frac{(1-p)-(1-p)^{\text{MAX}-\text{failed}+1}}{p}\Big)$ $+[\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed}$	$= \frac{(1-p)-(1-p)^{\text{MAX}+1}}{p}$

TABLE 5.1: Summary of the theoretical verification results for the expected number of loop iterations, probability of success, and the expected number of failed packets for `sendPacket`. The analysis is based on the pGCL model as defined in Listing 4.1. * Assuming that $p > 0$.

calc	g	**supremum** calc$[\![\texttt{while}_{\texttt{brp}}]\!](g)$	**preexpectation** calc$[\![\texttt{BRP}]\!](g)$
ert	$\mathbf{0}$	$\sqsubseteq \mathbf{1} + [\text{success}] \cdot \frac{1}{1-s}*$	$\sqsubseteq \mathbf{1} + \frac{1}{1-s}*$
wp	[success]	$= [\text{sent} < \text{N} \wedge \text{success}] \cdot s^{\text{N}-\text{sent}}$ $+[\text{sent} \geq \text{N} \wedge \text{success}]$	$= s^{\text{N}}$
wp	totalFailed	$= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Big($ $\text{totalFailed} + f \cdot \frac{1-s^{\text{N}-\text{sent}}}{1-s}\Big)$ $+[\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{totalFailed}$	$= f \cdot \frac{1-s^{\text{N}}}{1-s}$
wp	sent	$= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Big($ $\text{sent} + \frac{s(1-s^{\text{N}-\text{sent}})}{1-s}\Big)$ $+[\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent}$	$= \frac{s \cdot (1-s^{\text{N}})}{1-s}$

TABLE 5.2: Summary of the theoretical verification results for the expected number of loop iterations, probability of success, and the expected number of failed and sent packets for `BRP`. The analysis is based on the pGCL model as defined in Listing 4.2. * Assuming that $s < 1$.

5.5 Geometric Program

In subsequent sections of this thesis, it will be demonstrated that none of the computed preexpectations for the expected number of failed or sent packets could be verified. Although several possible explanations exist for this issue, the computed preexpectations were re-evaluated using probability theory [36] to verify their correctness.

As previously noted, the pGCL models `sendPacket` and `BRP` closely resemble a bounded geometric loop. Consequently, the expectations of these programs can be analysed similarly to the expectations of truncated geometric distributions. This section presents a probabilistic analysis of the expected number of trials and failures in a simple bounded geometric loop: a coin flip with success probability p, bounded by a maximum number of N flips.

5.5.1 Trials

For the number of trials to be less than N, the final trial must result in success. If the number of trials is exactly N, this implies that the first $(N-1)$ trials were unsuccessful, followed by a final trial, the outcome of which is irrelevant, as either it was successful or the maximum number of trials was reached. This results in the following probability mass function:

$$f(x) = \begin{cases} p \cdot (1-p)^{x-1} & 1 \leq x < N \\ (1-p)^{x-1} & x = N \\ 0 & x > N \end{cases}$$

Thus, the expected number of trials, $\mathbb{E}(X)$, is computed as follows:

$$\begin{aligned}
\mathbb{E}(X) &= \sum_{x=1}^{N} x \cdot f(x)) \\
&= \sum_{x=1}^{N-1} x \cdot f(x)_{x<N} + N \cdot f(x)_{x=N} \\
&= \sum_{x=1}^{N-1} x \cdot p \cdot (1-p)^{x-1} + N \cdot (1-p)^{N-1} \\
&= \sum_{x=1}^{N-1} x \cdot p \cdot \frac{(1-p)^x}{1-p} + N \cdot (1-p)^{N-1} \\
&= \frac{p}{1-p} \cdot \sum_{x=1}^{N-1} x \cdot (1-p)^x + N \cdot (1-p)^{N-1} \\
&= \frac{p}{1-p} \cdot \frac{1-p-N \cdot (1-p)^N + (N-1) \cdot (1-p)^{N+1}}{p^2} + N \cdot (1-p)^{N-1} \\
&= \frac{1-N \cdot (1-p)^{N-1} + (N-1) \cdot (1-p)^N + p \cdot N \cdot (1-p)^{N-1}}{p} \\
&= \frac{1-N \cdot (1-p) \cdot (1-p)^{N-1} + (N-1) \cdot (1-p)^N}{p} \\
&= \frac{1-N \cdot (1-p)^N + (N-1) \cdot (1-p)^N}{p}
\end{aligned}$$

... (continues on next page)

... (continued from previous page)

$$= \frac{1 - (1-p)^N}{p}$$

When applied to the `BRP` program, the number of trials corresponds to the number of additions to `totalFailed`, as this increment occurs in every loop iteration. However, rather than being incremented by one, `totalFailed` increases by f in each iteration. Consequently, the expected value of `totalFailed` should be the expected number of trials multiplied by f. Additionally, in `BRP`, the loop condition contains a negation of success rather than success. As a result, the probability of success is $(1-p)$ rather than p. Substituting p with the probabilistic variable s yields the expectation for the number of failures in `BRP`:

$$\mathbb{E}(\#\text{totalFailed}) = f \cdot \frac{1 - s^{\mathrm{N}}}{1 - s}$$

This equation is equivalent to the result obtained from the wp-calculations in Section 5.3.3. Therefore, the challenges encountered in verifying this property are unlikely to be the result of a computational error.

5.5.2 Failures

Similar to the expected number of trials, the expected number of failures can be computed using probability theory rather than wp-calculus. The probability mass function for the number of failures differs slightly from that of the number of trials, as failures are only incremented when the success condition is not met:

$$f(x) = \begin{cases} p \cdot (1-p)^x & 0 \leq x < N \\ (1-p)^x & x = N \\ 0 & x > N \end{cases}$$

Thus, the expected number of failures for the coin flip is given by:

$$\begin{aligned}
\mathbb{E}(X) &= \sum_{x=0}^{N} x \cdot f(x) \\
&= \sum_{x=0}^{N-1} x \cdot f(x)_{x<N} + N \cdot f(x)_{x=N} \\
&= \sum_{x=0}^{N-1} x \cdot p \cdot (1-p)^x + N \cdot (1-p)^N \\
&= p \cdot \sum_{x=0}^{N-1} x \cdot (1-p)^x + N \cdot (1-p)^N \\
&= p \cdot \frac{1 - p - N \cdot (1-p)^N + (N-1) \cdot (1-p)^{N+1}}{p^2} + N \cdot (1-p)^N \\
&= \frac{1 - p - N \cdot (1-p)^N + (N-1) \cdot (1-p)^{N+1} + p \cdot N \cdot (1-p)^N}{p} \\
&= \frac{1 - p - N \cdot (1-p)^{N+1} + (N-1) \cdot (1-p)^{N+1}}{p}
\end{aligned}$$

... (continues on next page)

... (continued from previous page)

$$= \frac{1 - p - (1-p)^{N+1}}{p}$$

When applied to `sendPacket`, the number of failures corresponds to the number of failed transmissions. Replacing the maximum number of trials N by the maximum number of failures MAX, the expected number of failed transmissions is given by:

$$\mathbb{E}(\#\text{failed}) = \frac{1 - p - (1-p)^{\text{MAX}+1}}{p}$$

which matched the computed preexpectation.

When relating the number of failures in the coin-flip model to `BRP`, a confusing correspondence emerges: the number of failures in the coin-flip model corresponds to the number of sent packets in `BRP`. Replacing the probability of a successful coin-flip ,p, with the probability of an unsuccessful sent packet, $(1-s)$, the expected number of sent packets is given by:

$$\mathbb{E}(\#\text{sent}) = \frac{s - s^{\text{N}+1}}{1 - s}$$

Again, this result is consistent with the computed preexpectation, further indicating that computational errors are unlikely to be responsible for the verification challenges.

CHAPTER 6

PRACTICAL VERIFICATION

This chapter translates the theoretical verification results from the previous chapter into practical verification using Caesar. It is structured as follows:

1. **Invariants:** The derivation of the loop invariants required for verification in Caesar.
2. **Translation from pGCL to HeyVL:** The process of converting the pGCL models into HeyVL code, incorporating a reward mechanism for the expected runtime and the implementation of exponentials.
3. **Results:** A summary of the outcomes of the practical verification using Caesar.

All code developed for this thesis is available for download from the accompanying repository [35].

6.1 Invariants

When verifying properties of programs containing a while-loop in Caesar, both upper and lower bounds of the preexpectation must be established separately. In this thesis, these bounds are verified via induction. To verify that a preexpectation is an upper bound, a *wp-superinvariant* is needed. Conversely, to verify a lower bound, a *wlp-subinvariant* must be provided.

For verifying PAST, upper bounds for the preexpectations of ert⟦`sendPacket`⟧(**0**) and ert⟦`BRP`⟧(**0**) were obtained using wp-superinvariants (see Sections 5.2.1 and 5.3.1). As a result, the translation of the verification of this property to Caesar required no extra steps.

In contrast, the theoretical verification of the remaining properties yielded exact expressions for the preexpectations (refer to Tables 5.1 and 5.2). Consequently, a wp-superinvariant and a wlp-subinvariant are needed to verify the upper and lower bounds, respectively. However, since it has been established that both `BRP` and `sendPacket` are PAST programs, the wp- and wlp-calculi are equivalent. Therefore, a wp-subinvariant also serves as a wlp-subinvariant:

property	program	upper bound	lower bound
termination	`sendPacket`	✓	N.A.
	`BRP`	✓	N.A.
success	`sendPacket`	✓	timeout
	`BRP`	timeout	✓
failures	`sendPacket`	timeout	timeout
	`BRP`	timeout	timeout
sent packets	`BRP`	timeout	timeout

TABLE 6.1: Caesar verification results for the manually computed preexpectations, using the computed suprema as wp-superinvariants and wlp-subinvariants. Refer to Tables 5.1 and 5.2 for the expressions of each preexpectation and supremum.

Theorem 7 (Equivalence wp- and wlp-subinvariants for an AST program). Let I be a wlp-subinvariant for the loop $\texttt{while}(\varphi)\{C\}$ of program P with respect to postexpectation f, and let P be AST (almost-surely terminating). Then:

$$\begin{aligned} & I \sqsubseteq \text{wlp}[\![\texttt{while}(\varphi)\{C\}]\!](f) && \text{(Theorem 5: coinduction)} \\ \wedge\ & \text{wp}[\![\texttt{while}(\varphi)\{C\}]\!](f) = \text{wlp}[\![\texttt{while}(\varphi)\{C\}]\!](f) && \text{(Theorem 3: wp v. wlp)} \\ \Longrightarrow\ & I \sqsubseteq \text{wp}[\![\texttt{while}(\varphi)\{C\}]\!](f) \end{aligned}$$

This result implies that coinduction can be applied using a wp-subinvariant, provided the program is AST. An analogous result holds for superinvariants.

This implies that the invariants derived through theoretical wp-analysis can be reused directly in Caesar to verify both upper and lower bounds. In particular, the suprema computed exactly for the probability of success, the expected number of failed transmissions, and the expected number of sent packets serve as both wp-superinvariants and wlp-subinvariants (see Appendix C for proofs). Therefore, they can theoretically be applied to verify both bounds of the corresponding preexpectations in Caesar.

Despite this theoretical applicability, the practical verification encountered challenges, as shown in Table 6.1. In response, alternative super- and subinvariants were explored to either verify the original preexpectations or establish alternative bounds. These alternative invariants, together with their corresponding properties, are listed in Table 6.2 and proven to be invariants in Appendix D. Notably, it was not possible to verify a non-trivial upper and lower bound for all properties across both programs using this approach.

6.2 From pGCL to HeyVL

Having determined the necessary equations for verification in Caesar, the next task involves translating the pGCL program into HeyVL. The programs provided in Listings 4.3 and 4.4 serve as initial references but must be modified according to the property that is being verified. Specifically, when verifying an upper bound, the appropriate `tag` is `wp`, or in

program	property	invariant	pre
`sendPacket`	failed	$[\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \Big($ $\text{failed} + (1-p)\cdot(\text{MAX} - \text{failed})\Big)$ $+[\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed}$	$(1-p)\cdot\text{MAX}$
`BRP`	success	$[\text{success}] \cdot s^{\text{N}-\text{sent}} + [\text{success}]$	s^{N}
`BRP`	failed	$[\text{sent} < \text{N} \wedge \text{success}] \cdot \Big($ $\text{totalFailed} + f\cdot(\text{N} - \text{sent})\Big)$ $+[\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed}$	$f \cdot \text{N}$
`BRP`	sent	$[\text{sent} < \text{N} \wedge \text{success}] \cdot \Big($ $\text{sent} + s\cdot(\text{N} - \text{sent})\Big)$ $+[\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{sent}$	$s \cdot \text{N}$

TABLE 6.2: Alternative invariants used for Caesar verification and their corresponding preexpectations. "Super" refers to a wp-superinvariant and a "sub" to a wlp-subinvariant.

the verification of termination, `ert`, and the `proc` must be substituted with a `coproc`. In contrast, for the verification of a lower bound, the relevant `tag` is `wlp`, and the `proc` remains unchanged. Moreover, the placeholders `preexp`, `postexp`, and `invar` must be substituted with the preexpectation, postexpectation, and invariant to be verified.

Further adjustments to the programs are required to ensure compatibility with the modified `ert`-calculus. Specifically, the statement `reward 1` needs to be incorporated within the body of each while-loop. Additionally, the assumptions necessary for the verification of PAST (specifically, $p > 0$ and $s < 1$) are introduced as axioms within the `Constants` domain.

6.2.1 Exponentials

The computed preexpectations and invariants for the probability of success, as well as for the expected number of failed transmissions and sent packets, introduce another challenge in the verification process using Caesar: the handling of exponentials. Since exponentiation is not natively supported in HeyVL, it must be implemented using a custom domain and axioms. Caution is required when working with axioms in Caesar, as any unsound axiom may lead to verification regardless of the correctness of the specified property and proof.

Listing 6.1 presents a recursive implementation of the `Exponentials` domain. In this implementation, line 2 specifies the function's type, line 3 defines the base case, and line 4 defines the inductive case.

LISTING 6.1: An recursive HeyVL Exponentials domain.

```
1 domain Exponentials {
2   func pow(num: UReal, power: UInt): UReal
3   axiom pow_base forall x: UReal. pow(x, 0) == 1
```

```
4   axiom pow_step forall x: UReal. forall y: UInt. pow(x, y + 1) ==
     x * pow(x,y)
5 }
```

Unfortunately, attempting to verify a program that incorporates this definition leads to timeout errors in Caesar. This issue can be explained by the work of Amin et al. [37], which describes how unrestricted recursion in an SMT-solver can result in non-termination or inefficiency due to matching loops or excessive instantiations. To mitigate this issue, the exponent is "unfolded", or "fuelled", by two steps to assist the SMT-solver. This approach, as presented in Listing 6.2, leaves `pow0` uninterpreted, thereby preventing the solver from entering infinite recursion, and allows controlled evaluation through single-step unfolding with `pow1`.

This two-step unfolding implementation is used for the exponential function `pow` throughout this thesis. Although additional unfolding steps were tested during the verification process, no improvements were observed compared to the two-step approach employed in Listing 6.2.

Listing 6.2: An fueled HeyVL Exponentials domain.

```
1 domain Exponentials {
2     func pow0(base: UReal, exponent: UInt): UReal
3     func pow1(base: UReal, exponent: UInt): UReal = ite(exponent ==
      0, 1, base * pow0(base, exponent - 1))
4     func pow(base: UReal, exponent: UInt): UReal = ite(exponent ==
     0, 1, base * pow1(base, exponent - 1))
5
6     axiom synonym_fuel1 forall b: UReal, e: UInt. pow(b, e) == pow1
     (b,e)
7 }
```

An alternative approach, which establishes a relationship between two exponentials by comparing their powers, is presented in Listing 6.3. This implementation is applicable exclusively to exponentials where the base is a probability ($0 \leq \text{base} \leq 1$), which aligns with the exponentials considered in this thesis. This approach facilitated faster verification of the upper bound of wp⟦`sendPacket`⟧([success]), demonstrating its potential. Specifically, it reduced the verification time to an average of 0.05 seconds, a significant improvement compared to the average of 5 seconds required when using the fuelled exponential domain. However, this method resulted in a timeout when verifying the lower bound of wp⟦`BRP`⟧([success]). Consequently, the implementation of `pow` shown in Listing 6.2 was adopted for the remainder of this thesis.

Listing 6.3: An alternative HeyVL Exponentials domain.

```
1 domain Exponentials {
2     func pow(base: UReal, exponent: UInt): UReal
3
4     axiom pow_order forall b: UReal, e1: UInt, e2: UInt.
5     (e1==e2 && pow(b, e1) == pow(b, e2)) || (e1>e2 && pow(b, e1) <
6     pow(b, e2)) || (e1<e2 && pow(b, e1) > pow(b, e2))
7 }
```

6.3 Results

The results of the practical verification using Caesar are summarised in Table 6.3.

Several additional notable observations were made during the execution of the Caesar verifications. In instances where two upper bounds were successfully verified, it is theoretically possible to implement the function call to `sendPacket` (as shown in Listing 4.5) within `BRP` and verify the upper bound without the use of the representative variables s and f. However, attempts to do so resulted in timeout errors in Caesar, both under the shorter timeout of 20 seconds and the standard timeout of 300 seconds.

Another significant observation arose during the Caesar verification process: when multiple procedures were executed within a single file, certain combinations caused Caesar to time out. This phenomenon is further discussed in the Discussion chapter of this thesis (Chapter 7).

property	**program**	**computed preexpectation**	**verified upper bound**	**verified lower bound**	**assume**
termination	`sendPacket`	$\sqsubseteq \mathbf{1} + \frac{1}{p}$	$1 + \frac{1}{p}$	N.A.	$p > 0$
	`BRP`	$\sqsubseteq \mathbf{1} + \frac{1}{1-s}$	$1 + \frac{1}{1-s}$	N.A.	$s < 1$
success	`sendPacket`	$1 - (1-p)^{\mathrm{MAX}}$	$1 - (1-p)^{\mathrm{MAX}}$	-	
	`BRP`	s^{N}	s^{N}	s^{N}	
failures	`sendPacket`	$\frac{(1-p)-(1-p)^{\mathrm{MAX}+1}}{p}$	MAX	-	
	`BRP`	$f \cdot \frac{1-s^{\mathrm{N}}}{1-s}$	$f \cdot \mathrm{N}$	-	
sent packets	`BRP`	$\frac{s-s^{\mathrm{N}+1}}{1-s}$	$s \cdot \mathrm{N}$	-	

TABLE 6.3: An overview of the manually computed preexpectations, the upper and lower bounds verified using Caesar, and any necessary assumptions for each property. A dash indicates that only a trivial upper or lower bound was verified.

CHAPTER 7

DISCUSSION

This chapter addresses the main research question and its sub-questions, introduced in Section 1, following the stepwise verification process outlined in Chapters 5 and 6. Each sub-question is discussed individually, followed by a discussion of the main research question.

7.1 BRP Abstraction

How can BRP be modelled to facilitate the verification using Caesar?

Since Caesar is based on the weakest preexpectations calculus, BRP must be represented in probabilistic Guarded Command Language (pGCL) to enable manual reasoning about its behaviour.

Two pGCL models of BRP were developed in this study. The first model contains a single while-loop with two probabilistic choices. The second model decomposes the protocol into two separate programs, each containing a while-loop and a probabilistic choice.

Applying fixed-point iteration to the loop-characteristic function of the first model did not yield a supremum. This difficulty does not stem from a flaw in Caesar or in the BRP model itself, but rather reflects a general limitation of the weakest preexpectation calculus. Identifying a supremum heavily depends on the ability of the verifier to recognise useful patterns, once again highlighting the inherent difficulty of the verification process. The absence of errors in the computed equations for Φ^1 through Φ^4 was confirmed using Caesar's loop-unrolling feature. While this feature was helpful, it has inherent limitations: it can refute candidate upper bounds for the least fixed point but cannot provide definitive correctness guarantees.

The construction of the second, decomposed model proved particularly helpful in the verification process. Its resemblance to a geometric program enabled cross-verification of manual calculations and theoretical results. I should note, however, that the inversion in the naming of "failures" between the geometric program and the proposed model initially led me to second-guess of what turned out to be correct calculations. This experience underscored the importance of careful translation of program variables. When such translations are handled with care, abstractions of this kind can serve as highly effective tools in the verification process.

Although insights gained from verifying the second model could potentially be used to revisit the first model, doing so would not align with the primary goal of this research, which is to evaluate the verification capabilities of Caesar rather than verify BRP itself. The abstraction of BRP into two geometric-like programs aligns well with this objective

and has already introduced sufficient challenges and insights. A more complex model may become useful for further testing once these challenges have been addressed.

7.2 Theoretical Verification

> *Which properties of BRP can be verified using Caesar's underlying weakest preexpectation calculus?*

Once an appropriate pGCL model of BRP was established, the following properties were selected for verification:

1. Positive almost-sure termination (PAST)
2. Probability of success
3. Expected number of failed transmissions
4. Expected number of sent packets

Properties 2 through 4 each correspond to a key variable in the BRP model. These properties also reflect typical goals in probabilistic program analysis: the calculation of expected runtimes, expected values of program variables, and the probabilities of certain outcomes.

Establishing PAST was a crucial first step, as it permits the interchangeable use of weakest preexpectation (wp) and weakest liberal preexpectation (wlp) calculus. Furthermore, the complexity of verification was deliberately increased over the course of the study. Beginning with PAST was therefore a pragmatic and strategic choice.

To verify PAST, two assumptions were introduced: $p < 1$ and $s > 0$. These are justified in Sections 5.2.1 and 5.3.1 by arguing that BRP would not be applicable in scenarios where these conditions do not hold.

With the experience I have gained through this thesis, I believe it would be feasible to revisit the verification of PAST without relying on these assumptions. However, as previously argued, they are reasonable within the context of BRP, and given time constraints, further refinement was not pursued.

Another property of interest, not verified in this study, is the overall expected runtime. While the verification of PAST confirms that the number of loop iterations is finite, it does not establish anything about the overall expected runtime. However, given that ert-calculus was already employed in the PAST analysis, verifying runtime is unlikely to yield novel insights into Caesar's verification capabilities. The selected properties already encapsulate the key questions in probabilistic program analysis while systematically increasing the complexity of verification in Caesar.

In my experience, the most technically challenging aspect of the theoretical verification was handling summations where the guard depended on the summation index. This initially proved difficult, but with practice, I was able to develop effective simplification strategies. These techniques are outlined in the recommendations provided in Subsection 7.6.

7.3 Translation Steps

> *How can theoretical verification based on weakest preexpectations be translated into verification using Caesar?*

The most significant differences between theoretical verification using weakest preexpectations and practical verification using Caesar in this case study pertain to the handling of loops. In particular, Caesar requires the application of a proof rule, something that is not necessary in theoretical verification. Moreover, when applying Caesar's proof rules, such as induction, loop unrolling, and ω-invariants, one must separately verify both upper and lower bounds for the desired preexpectation. In contrast, theoretical wp-calculus often yields an exact value directly.

Although this study primarily focuses on the induction proof rule, the following two steps are broadly applicable in translating any theoretical wp-based verification into a Caesar-based approach and are further elaborated in this section:

1. Selection of an appropriate proof rule.
2. Translation of the pGCL program into a HeyVL program.

7.3.1 Proof Rules

Two proof rules were applied in this study: induction and loop unrolling. Loop unrolling was used in a specific context: to validate the correctness of fixed-point iterations. This particular application is described in Section 5.1.1, which also provides a reusable template for this technique.

Induction was applied more extensively throughout the study. It requires specifying a loop invariant, and determining the appropriate combination of procedure structure and invariant can be quite challenging, even with the help of Table 2.4. To support this process, Figure 7.1 presents a flowchart designed to guide users in applying the induction rule in Caesar, given a program, preexpectation, and postexpectation. The resulting structure can then be applied using the HeyVL procedure format specified in Definition 16.

Definition 16 (HeyVL procedure using induction).

```
proc P (in : τ) -> (out : τ)
    pre ϕ
    post ψ
{
    S1
    @invariant(I)
    while (φ) {
        Sbody
    }
    S2
}
```

Where $\overline{in}$ and $\overline{out}$ denote lists of typed read-only inputs and outputs. $\texttt{S}_1$ and $\texttt{S}_2$ represent HeyVL statements preceding and following the loop, respectively, while $\texttt{S}_{\texttt{body}}$ defines the loop body. The loop invariant is denoted by I, the loop guard by φ, the preexpectation by ϕ, and the postexpectation by ψ.

By definition, the supremum obtained from theoretical wp-calculus serves as both a superinvariant and subinvariant within the same calculus, as demonstrated by the computations in Appendix C. Furthermore, if a program satisfies almost-sure termination

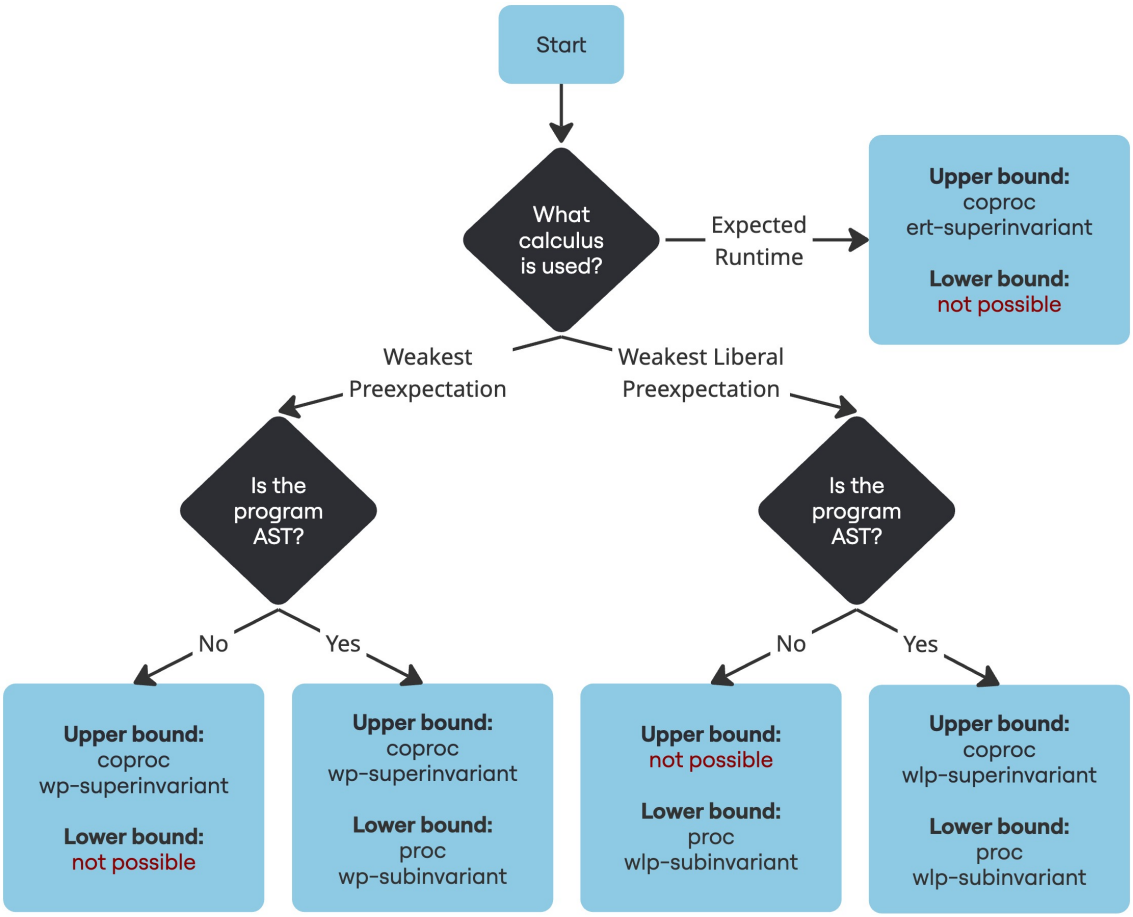

FIGURE 7.1: A flowchart describing the process of determining the appropriate HeyVL procedure and loop invariant, assuming the use of the induction proof rule.

(AST), wp- and wlp-calculus become equivalent, meaning that wp-subinvariants and wlp-subinvariants (as well as superinvariants) are interchangeable. This equivalence is formally proven in Section 6.1.

In this study, theoretical verification employed wp-calculus, and the program satisfies positive almost-sure termination (PAST), which implies AST. Thus, in theory, the supremum derived from the while-loop should serve as a valid loop invariant when verifying upper and lower bounds using induction in Caesar. However, in practice, this approach did not always succeed, a point discussed further in Section 7.5.

If, as was the case in this study, Caesar is unable to verify a preexpectation using the computed supremum as an invariant, it may be necessary to identify an alternative loop invariant. This process is outlined in Figure 7.2. Once a new invariant is established, a corresponding preexpectation must be computed. This is likely to yield a weaker preexpectation, although in favourable cases, one may find an alternative invariant that still supports verification of the original preexpectation.

7.3.2 pGCL to HeyVL

This part of the translation process was relatively straightforward. In my experience, translating a pGCL program into HeyVL is intuitive, largely due to the structural and syntactic similarities between the two languages. In cases where the translation was not

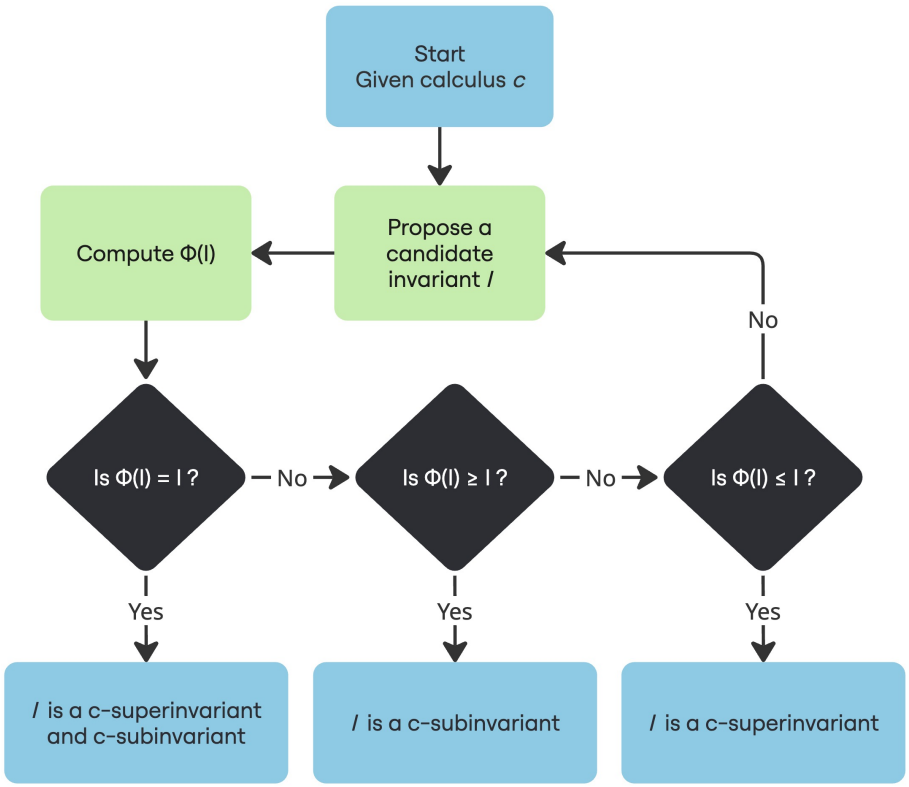

FIGURE 7.2: A flowchart describing the process of determining a loop invariant.

immediately clear, or when I needed a refresher on the notation, the examples provided on the Caesar website and the linked GitHub repository [26] were helpful. However, these resources are somewhat scattered, and locating the most relevant example can be time-consuming.

A suggestion for future work that emerged from this experience is the completion of the "A Zoo of HeyVL Examples" page on the Caesar website. A well-structured and comprehensive collection of examples would significantly streamline the translation process for future users.

7.4 Translation Observations

> *What challenges and insights emerge when translating the theoretical verification to practical verification using Caesar?*

The two translation steps outlined in the previous section are conceptually clear: selecting a suitable proof rule depends on the verification method, and the structural resemblance between pGCL and HeyVL makes program translation relatively straightforward. However, a major challenge arose in encoding exponentials using axioms in HeyVL.

An important pitfall is that unsound axioms can cause Caesar to report successful verification regardless of a procedure's content, preexpectation, and postexpectation. At one point during this project, I unintentionally introduced such an unsound axiom while attempting to encode exponentials. This led me to falsely conclude that a property had been successfully verified. The issue became apparent only when, after accidentally saving a version of the program with an incorrect preexpectation, the verification still succeeded. To avoid this problem going forward, I adopted a strategy of sanity-checking each axiom

by attempting to verify an intentionally incorrect bound. If Caesar accepted this bound, the axiom was deemed unsound. This difficulty lies not in Caesar itself, but in the human process of correctly formulating sound axioms.

The initial sound recursive definition of exponentials, shown in Listing 6.1, posed a different challenge. Although it was semantically correct, it caused the SMT-solver underlying Caesar to get stuck. Unlike the previous issue, this limitation originates from Caesar's current implementation. Ongoing parallel research is addressing this limitation, but for the purposes of this thesis, a fuelled definition of exponentials was used to support partial verification. While this workaround alleviated the problem in some cases, it did not resolve all verification challenges. Further discussion on this topic is provided in Section 7.5.

Another strategy for mitigating the challenges posed by exponentials involved identifying simplified invariants that still enabled verification. This approach was successfully applied, for example, in verifying a lower bound for the probability of success in `BRP`. However, this workaround introduced a new difficulty: identifying a suitable invariant in the first place. Whilst verifying whether a candidate invariant is a superinvariant or subinvariant is relatively straightforward, determining a candidate invariant lacks a systematic methodology and relies heavily on insight and experimentation.

Interestingly, recent work by Batz et al. [29] introduces a tool, cegispro2, which supports the synthesis of sub- and superinvariants for probabilistic programs. This tool may offer a promising solution to the problem of invariant discovery and is discussed in more detail in Section 9.

7.5 Practical Verification Results

> *What are the verification results, and what assumptions, if any, are required to verify each property?*

The verification results are summarised in Table 6.3 on page 49.

The first notable observation concerns the verification of the probability of success. Initially, the loop invariant was set to the supremum derived from the theoretical analysis, as shown in Tables 5.1 and 5.2. This approach successfully verified the upper bound for both programs, but the same expression could not be verified as a lower bound.

For the expected number of failed and sent transmissions, verification proved even more challenging: the computed preexpectations could not be verified as either upper or lower bounds. These difficulties are attributable not to modelling or human error, but to limitations within Caesar itself. In each case, verification failed due to SMT-solver timeouts, not counterexamples, suggesting that the solver struggled with the exponentials involved. The additional presence of exponentials within fractions may have exacerbated the problem. This could explain why verification success decreased with increasing complexity of the expectations, though this remains speculative.

To circumvent these issues, alternative loop invariants were explored. For the upper bound of the probability of success in `BRP`, a simpler invariant was found that allowed successful verification. However, no such alternative was identified for `sendPacket`. For the last two properties, only non-trivial superinvariants could be found, enabling the verification of non-trivial upper bounds. Unfortunately, no suitable non-trivial subinvariants could be determined, leaving the lower bounds unverified.

Another unexpected issue emerged with function calls: versions of `BRP` containing calls to `sendPacket` consistently failed to verify. Replacing placeholder variables with function calls, even with properly adjusted preexpectations and invariants, resulted in timeouts. The cause of this behaviour remains unclear.

Finally, verification outcomes appeared to depend on the grouping of procedures within a file. The non-trivial upper bound for `BRP`'s probability of success and expected number of failures failed to verify in isolation, resulting in a timeout, but successfully verified when included alongside any procedure for `sendPacket`. Similarly, verifying only the non-trivial upper bound of the expected number of sent packets took 0.39 seconds, whereas adding a trivial lower bound procedure reduced the verification time to 0.11 seconds. The reason for these variations remains unknown.

7.6 A Guide to Caesar

> *What are the advantages and shortcomings of Caesar's verification capabilities when applied to the Bounded Retransmission Protocol (BRP), and what recommendations for its application can be derived from this case study?*

7.6.1 Advantages

In my opinion, the primary advantage of Caesar lies in its potential to reduce human error in the verification process. It facilitates rapid testing of theoretical preexpectations, enabling users to explore their hypotheses through tool support rather than relying entirely on manual computations. Ideally, Caesar would allow non-experts to carry out weakest preexpectation-style verification. For simple programs without loops, I believe this is already achievable. However, as long as inductive proofs require the manual formulation of loop invariants, a certain level of expertise remains necessary.

I also see value in Caesar as a tool to support, rather than replace, theoretical verification. For example, I used loop unrolling to check my fixed-point iteration calculations. With more testing and case studies, I expect additional applications of Caesar to emerge.

7.6.2 Limitations

That said, several limitations became apparent during my work. Caesar struggles with verifying preexpectations involving exponentials, especially when they occur inside fractions, which frequently leads to SMT-solver timeouts. Function calls in programs also posed a challenge: replacing a verified procedure with a function call to the same code caused verification to fail.

Even more puzzling was Caesar's unpredictable behaviour depending on how procedures are grouped. In some cases, verification failed when procedures were isolated, but succeeded when unrelated ones were included in the same file. While I was able to work around this, it undermines the reliability of results and complicates the workflow.

Finally, limitations from the theoretical approach carry over. The need for inductive invariants is a substantial hurdle. Ideally, a verification tool like Caesar would relieve the user of this burden, but at present, that is not the case. Ongoing research on probabilistic invariant synthesis, such as the work by Batz et al. [29], may help address this challenge in the future (see Section 9).

7.6.3 Recommendations

Based on my experience, I offer the following practical recommendations for working with Caesar.

Rewrite programs to resemble geometric loops

Where possible, I recommend rewriting programs to resemble geometric loops, as was done throughout this thesis. Even if this results in an abstraction of the original program, it often simplifies reasoning about preexpectations and provides a solid foundation for inductive verification.

Use placeholder variables for function calls

When abstracting programs, function calls may arise, which can complicate reasoning about expected behaviour. I found it helpful to replace these calls with placeholder variables that represent their outcomes. This simplifies both the theoretical reasoning and the Caesar specification. If necessary, these placeholders can later be substituted with concrete expressions, assuming Caesar's handling of function calls improves in the future.

Use loop-unrolling to check fixed-point iteration

If you're stuck on a fixed-point iteration, it can be helpful to use Caesar to check your work rather than repeatedly reworking the equations by hand. This can be done through loop unrolling, as described in detail in Section 5.1.1.

Simplify guarded summations

When simplifying guarded summations, I found the following simplifications especially helpful:

Definition 17 (Equations for the simplification of sums).
Let $k, n \in \mathbb{N}$, and suppose x and a are expressions not involving i, while $f(i)$ contains i. Then:

$$\sum_{i=k}^{\infty} [i = x] \cdot f(i) = f(x). \tag{7.1}$$

$$\sum_{i=k}^{\infty} [i < x] \cdot f(i) = \sum_{i=k}^{x-1} f(i) \text{ iff } k \leq x - 1. \tag{7.2}$$

If $k = 0$, one of the following geometric sum formulas may apply:

$$\sum_{i=0}^{n-1} i \cdot r^i = \frac{r - n \cdot r^n + (n-1) \cdot r^{n+1}}{(1-r)^2} \text{ iff } |r| < 1. \tag{7.3}$$

$$\sum_{i=0}^{n} a \cdot r^i = \begin{cases} a \cdot (n+1) & r = 1, \\ a \cdot \frac{1-r^{n+1}}{1-r} & |r| < 1. \end{cases} \tag{7.4}$$

$$\sum_{i=0}^{n} (1-r) \cdot a \cdot r^i = a \cdot (1 - r^{n+1}) \text{ iff } |r| \leq 1. \tag{7.5}$$

Verify PAST instead of AST using a modified `ert`-calculus

Although the verification of AST (Almost-Sure Termination) can simplify the overall verification process (see Figure 7.1), I recommend starting with the verification of PAST (Positive Almost-Sure Termination), particularly using a modified `ert`-calculus, as was done in this thesis. Since PAST implies AST and is significantly easier to prove with this approach, it provides a more accessible entry point into the verification process.

Lower the timeout limit

Finally, during exploratory work, I found it helpful to lower the SMT-solver timeout from the default to 20 seconds. This significantly reduced idle time and allowed me to test more variants quickly: a small but effective improvement to the workflow.

CHAPTER 8

CONCLUSION

This thesis set out to evaluate the verification capabilities of Caesar by applying it to the Bounded Retransmission Protocol (BRP). A stepwise verification strategy was adopted, beginning with a theoretical analysis using the weakest preexpectation calculus, followed by practical verification using Caesar. Throughout this process, several challenges were encountered and addressed, offering insights into both the strengths and weaknesses of Caesar and informing recommendations for its effective use.

A key contribution of this thesis is the abstraction and decomposition of BRP into two geometric-like programs, which enabled more effective reasoning about the protocol's behaviour and facilitated the stepwise verification process. The theoretical verification of key properties (positive almost-sure termination, success probability, and the expectation of the number of failed and sent transmissions) was largely successful. These properties reflect the central questions in probabilistic program analysis while systematically increasing the complexity of verification.

Translating the results of the theoretical analysis into formal verification using Caesar introduced new challenges. In contrast to the manual wp-calculus, Caesar requires explicit application of proof rules and inductive reasoning, introducing a new layer of complexity. Specifically, this highlighted the challenge of invariant discovery, a process that remains largely manual and experimental. Although supremum expressions derived theoretically often constitute valid invariants, these did not always yield successful verification in practice due to limitations in Caesar's current implementation, particularly regarding SMT-solver performance and the handling of exponentials.

Despite these hurdles, the case study revealed several strengths of the tool. Loop unrolling proved useful for validating manually computed fixed-point iterations, and Caesar allowed for rapid testing through trial and error. With certain workarounds, meaningful properties of BRP were successfully verified. Still, the study identified several areas in need of improvement, including support for recursive axiom definitions, improving handling of function calls, and the unexplained impact of procedure groupings on verification outcomes.

In conclusion, Caesar demonstrated considerable potential as a tool for the verification of probabilistic protocols such as BRP. To fully realise this potential, however, several technical and usability issues must be addressed. The recommendations and observations presented throughout this thesis, including proposed flowcharts and verification templates, aim to support future users in navigating Caesar-based verification more effectively. Continued development of Caesar will be essential in broadening its capabilities, and directions for future work are outlined in Chapter 9.

CHAPTER 9

FUTURE WORK

The future work stemming from this study includes addressing existing issues in Caesar, exploring potential improvements to the tool, and conducting further evaluation through additional case studies.

9.1 Address Existing Issues

Several limitations identified in Section 7.6 this study point to important directions for future work. Specifically, three unresolved issues merit attention: the SMT-solver getting stuck when handling recursive axioms, the use of function calls resulting in failed verification, and the unexplained influence of procedure groupings on verification outcomes. While the first issue is currently being explored in a parallel study, the latter two problems remain unaddressed. These fundamental problems should be resolved before pursuing other suggestions outlined in this chapter.

Addressing these issues will likely require further investigation and testing to isolate and understand their causes. Given my limited insight into Caesar's internal implementation, the solution to these issues remains unclear.

9.2 Tool Improvements

Once the aforementioned issues have been resolved, several improvements may be made to enhance Caesar's usability. These suggestions aim to support both new and experienced users.

First, the Caesar documentation would benefit from the addition of structured HeyVL examples. The inclusion of such examples would support users in encoding common patterns and applying verification strategies, eliminating the need to construct these from scratch each time. This would be especially valuable for those unfamiliar with Caesar and HeyVL.

Second, Caesar's error messaging could be improved. In early use of the tool, it was often difficult to distinguish between errors caused by invariants and those due to mistakes in the preexpectation. A clearer classification of error messages, perhaps accompanied by documentation or examples, would make debugging much easier.

A further, more ambitious improvement would be the automation of inductive invariant generation. Tools such as cegispro2 [29] suggest that this may be possible. If Caesar could automatically synthesise inductive invariants, the tool would become far more accessible to users without expertise in weakest preexpectation-style verification. While likely

challenging to implement, this represents an exciting long-term goal, though it is not an immediate priority.

9.3 Further Evaluation

In addition to addressing existing issues and improving usability, further empirical evaluation of Caesar constitutes an important avenue of future work. Once support for limited function has been fully implemented, the present study may be revisited, particularly to make use of the recursive definition of exponentials given in Section 6.2.1, which was not usable in the present research.

Moreover, it may be valuable to verify increasingly complex and realistic models of the Bounded Retransmission Protocol. This would test Caesar's practical capabilities and may also allow for the application of advanced proof rules not used in this study, such as ω-Invariants, Almost-Sure Termination, and the Optional Stopping Theorem.

Finally, further case studies could be explored to test Caesar's applicability across a broader range of probabilistic programs. However, given the substantial scope of identified future work, I consider this a lower priority at present.

BIBLIOGRAPHY

[1] O. Abril-Pla et al. "PyMC: a modern, and comprehensive probabilistic programming framework in Python". In: *PeerJ Computer Science* 9 (Sept. 2023), e1516. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.1516. URL: https://peerj.com/articles/cs-1516.

[2] E. Bingham et al. "webppl-oed: A practical optimal experiment design system". In: *escholarship.org* 20 (2019), pp. 1–6. URL: https://escholarship.org/uc/item/1tq428hv.

[3] B. Carpenter et al. "Stan: A Probabilistic Programming Language". In: *Journal of Statistical Software* 76.1 (Jan. 2017), pp. 1–32. ISSN: 1548-7660. DOI: 10.18637/jss.v076.i01. URL: http://www.jstatsoft.org/v76/i01/.

[4] A. D. Gordon et al. "Probabilistic programming". In: *Future of Software Engineering Proceedings*. New York, NY, USA: ACM, May 2014, pp. 167–181. ISBN: 9781450328654. DOI: 10.1145/2593882.2593900. URL: https://dl.acm.org/doi/10.1145/2593882.2593900.

[5] N. Arora, S. Russell, and E. Sudderth. "NET-VISA: Network Processing Vertically Integrated Seismic Analysis". In: *Bulletin of the Seismological Society of America* 103.2A (Apr. 2013), pp. 709–729. ISSN: 0037-1106. DOI: 10.1785/0120120107.

[6] D. J. Fremont et al. "Scenic: a language for scenario specification and scene generation". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, June 2019, pp. 63–78. ISBN: 9781450367127. DOI: 10.1145/3314221.3314633. URL: https://dl.acm.org/doi/10.1145/3314221.3314633.

[7] A. McIver and C. Morgen. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. New York: Springer-Verlag, 2005. ISBN: 0-387-40115-6. DOI: 10.1007/b138392. URL: http://link.springer.com/10.1007/b138392.

[8] V. D'Silva, D. Kroening, and G. Weissenbacher. "A Survey of Automated Techniques for Formal Software Verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (July 2008), pp. 1165–1178. ISSN: 0278-0070. DOI: 10.1109/TCAD.2008.923410. URL: http://ieeexplore.ieee.org/document/4544862/.

[9] P. Beynon-Davies. "Information systems 'failure': the case of the London Ambulance Service's Computer Aided Despatch project". In: *European Journal of Information Systems* 4.3 (Aug. 1995), pp. 171–184. ISSN: 0960-085X. DOI: 10.1057/ejis.1995.20. URL: https://www.tandfonline.com/doi/full/10.1057/ejis.1995.20.

[10] A. Y. Xu et al. "Failure modes and effects analysis (FMEA) for Gamma Knife radiosurgery". In: *Journal of Applied Clinical Medical Physics* 18.6 (Nov. 2017), pp. 152–168. ISSN: 1526-9914. DOI: 10.1002/acm2.12205. URL: https://aapm.onlinelibrary.wiley.com/doi/10.1002/acm2.12205.

[11] E. W. Dijkstra. "The humble programmer". In: *Communications of the ACM* 15.10 (Oct. 1972), pp. 859–866. ISSN: 0001-0782. DOI: 10.1145/355604.361591. URL: https://dl.acm.org/doi/10.1145/355604.361591.

[12] B. L. Kaminski. "Advanced Weakest Precondition Calculi for Probabilistic Programs". PhD thesis. RWTH Aachen University, Feb. 2019. DOI: 10.18154/RWTH-2019-01829. URL: https://publications.rwth-aachen.de/record/755408/files/755408.pdf.

[13] E. W. Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs". In: *Communications of the ACM* 18.8 (Aug. 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975. URL: https://dl.acm.org/doi/10.1145/360933.360975.

[14] L. Helmink, M. P. A. Sellink, and F. W. Vaandrager. "Proof-checking a data link protocol". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 806 LNCS. Springer, Berlin, Heidelberg, 1994, pp. 127–165. DOI: 10.1007/3-540-58085-9_75. URL: http://link.springer.com/10.1007/3-540-58085-9_75.

[15] P. Schröer et al. "A Deductive Verification Infrastructure for Probabilistic Programs". In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA2 (Oct. 2023), pp. 2052–2082. ISSN: 2475-1421. DOI: 10.1145/3622870. URL: https://dl.acm.org/doi/10.1145/3622870.

[16] P. R. D'Argenio et al. "Reachability Analysis of Probabilistic Systems by Successive Refinements". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 2165. Springer, Berlin, Heidelberg, 2001, pp. 39–56. DOI: 10.1007/3-540-44804-7_3. URL: http://link.springer.com/10.1007/3-540-44804-7_3.

[17] B. L. Kaminski, J.-P. Katoen, and C. Matheja. "On the hardness of analyzing probabilistic programs". In: *Acta Informatica* 56.3 (Apr. 2019), pp. 255–285. ISSN: 0001-5903. DOI: 10.1007/s00236-018-0321-1. URL: http://link.springer.com/10.1007/s00236-018-0321-1.

[18] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall series in automatic computation. Prentice-Hall, 1976. ISBN: 9780132158718.

[19] S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum. *Handbook of Logic in Computer Science: Volume 3: Semantic Structures*. Clarendon Press, 1995, p. 512. ISBN: 9780198537625.

[20] J.-L. Lassez, V. Nguyen, and E. Sonenberg. "Fixed point theorems and semantics: a folk tale". In: *Information Processing Letters* 14.3 (May 1982), pp. 112–116. ISSN: 00200190. DOI: 10.1016/0020-0190(82)90065-5. URL: https://linkinghub.elsevier.com/retrieve/pii/0020019082900655.

[21] R. W. Floyd. "Assigning Meanings to Programs". In: *Mathematical Aspects of Computer Science*. Springer, Dordrecht, 1993, pp. 65–81. DOI: 10.1007/978-94-011-1793-7_4. URL: http://link.springer.com/10.1007/978-94-011-1793-7_4.

[22] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://dl.acm.org/doi/10.1145/363235.363259.

[23] D. Park. "Fixpoint Induction and Proofs of Program Properties". In: *Machine Intelligence 5* (1969). URL: https://cir.nii.ac.jp/crid/1573950399497019904.

[24] B. L. Kaminski et al. "Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9632. Springer Verlag, 2016, pp. 364–389. DOI: 10.1007/978-3-662-49498-1_15. URL: http://link.springer.com/10.1007/978-3-662-49498-1_15.

[25] K. Batz et al. "J-P: MDP. FP. PP". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 15260 LNCS. Springer Science and Business Media Deutschland GmbH, 2025, pp. 255–302. DOI: 10.1007/978-3-031-75783-9_11. URL: https://link.springer.com/10.1007/978-3-031-75783-9_11.

[26] *Docs | Caesar*. URL: https://www.caesarverifier.org/docs (visited on 02/19/2025).

[27] P. R. D'Argenio et al. "The bounded retransmission protocol must be on time!" In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 1217. Springer, Berlin, Heidelberg, 1997, pp. 416–431. DOI: 10.1007/BFb0035403. URL: http://link.springer.com/10.1007/BFb0035403.

[28] J. Spel. *Monotonicity in Markov chains*. May 2018. URL: http://essay.utwente.nl/74981/.

[29] K. Batz et al. "Probabilistic Program Verification via Inductive Synthesis of Inductive Invariants". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 13994 LNCS. Springer Science and Business Media Deutschland GmbH, 2023, pp. 410–429. DOI: 10.1007/978-3-031-30820-8_25. URL: https://link.springer.com/10.1007/978-3-031-30820-8_25.

[30] C. Paulin-Mohring. "Introduction to the Coq Proof-Assistant for Practical Software Verification". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7682 LNCS. Springer, Berlin, Heidelberg, 2012, pp. 45–95. DOI: 10.1007/978-3-642-35746-6_3. URL: http://link.springer.com/10.1007/978-3-642-35746-6_3.

[31] G. Holzmann. *Design and validation of computer protocols.(1991)*. Prentice-Hall, 1991. ISBN: 0-13-539834-7.

[32] J. Bengtsson et al. "UPPAAL — a tool suite for automatic verification of real-time systems". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 1066. Springer, Berlin, Heidelberg, 1996, pp. 232–243. DOI: 10.1007/BFb0020949. URL: http://link.springer.com/10.1007/BFb0020949.

[33] D'Argenio et al. *PRISM - Case Studies - Bounded Retransmission Protocol*. URL: https://www.prismmodelchecker.org/casestudies/brp.php (visited on 10/25/2024).

[34] K. Batz et al. "Latticed k-Induction with an Application to Probabilistic Programs". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Ed. by A. Silva and K. R. M. Leino. Vol. 12760. Lecture Notes in Computer Science. Cham: Springer International Publishing, May 2021, pp. 524–549. DOI: 10.1007/978-3-030-81688-9_25. URL: https://link.springer.com/10.1007/978-3-030-81688-9_25.

[35] F. van Jaarsveld. *Code Accompanying Master's Thesis on Practical Probabilistic Program Verification using Caesar*. DOI: 10.5281/zenodo.15408742. URL: https://zenodo.org/records/15408743.

[36] D. Bertsekas and J. Tsitsiklis. *Introduction to probability*. Athena Scientific, 2008. ISBN: 978-1-886529-23-6. URL: http://athenasc.com/probbook.html.

[37] N. Amin, K. R. M. Leino, and T. Rompf. "Computing with an SMT Solver". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 8570 LNCS. Springer, Cham, 2014, pp. 20–35. DOI: 10.1007/978-3-319-09099-3_2. URL: http://link.springer.com/10.1007/978-3-319-09099-3_2.

APPENDIX A

FIXED-POINT ITERATION

This appendix presents the fixed-point iteration equations corresponding to the loop-characteristic functions in Chapter 5. For better readability, the notation $\Phi^n(\mathbf{0})$ is simplified to Φ^n.

A.1 Initial Attempt

This section shows the fixed-point iteration for the while-loop of $\mathrm{wp}[\![P]\!](1)$, where P denotes the initial pGCL model of BRP as shown in Listing 5.1 on page 30. The corresponding verification attempt is discussed in detail in Section 5.1.

Loop-characteristic function

$$\Phi(X) = [\text{sent} < \text{N} \wedge \text{failed} < \text{MAX}] \cdot \mathrm{wp}[\![\texttt{body}]\!](X) + [\text{sent} \geq \text{N} \vee \text{failed} \geq \text{MAX}]$$

$$\mathrm{wp}[\![\text{body}]\!](X) = p^2 \cdot X[\text{sent} := \text{sent} + 1; \text{failed} := 0] + (1+p) \cdot (1-p) \cdot X[\text{failed} := 0]$$

Fixed-point iteration

$$\begin{aligned}
\Phi^1 &= [\text{sent} \geq \text{N} \vee \text{failed} \geq \text{MAX}] \\
\Phi^2 &= p^2 \cdot [\text{sent} + 1 = \text{N} \wedge \text{failed} < \text{MAX}] \\
&\quad + (1+p) \cdot (1-p) \cdot [\text{sent} < \text{N} \wedge \text{failed} + 1 = \text{MAX}] \\
&\quad + [\text{sent} \geq \text{N} \vee \text{failed} \geq \text{MAX}] \\
\Phi^3 &= p^4 \cdot [\text{sent} + 2 = \text{N} \wedge \text{failed} < \text{MAX}] \\
&\quad + p^2 \cdot [\text{sent} + 1 = \text{N} \wedge \text{failed} < \text{MAX}] \\
&\quad + p^2 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 1 < \text{N} \wedge \text{failed} < \text{MAX} \wedge \text{MAX} = 1] \\
&\quad + p^2 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 1 = \text{N} \wedge \text{failed} + 1 < \text{MAX}] \\
&\quad + (1+p) \cdot (1-p) \cdot [\text{sent} < \text{N} \wedge \text{failed} + 1 = \text{MAX}] \\
&\quad + (1+p)^2 \cdot (1-p)^2 \cdot [\text{sent} < \text{N} \wedge \text{failed} + 2 = \text{MAX}] \\
&\quad + [\text{sent} \geq \text{N} \vee \text{failed} \geq \text{MAX}]
\end{aligned}$$

$$
\begin{aligned}
\Phi^4 = {} & p^6 \cdot [\text{sent} + 3 = \text{N} \wedge \text{failed} < \text{MAX}] \\
& + p^4 \cdot [\text{sent} + 2 = \text{N} \wedge \text{failed} < \text{MAX}] \\
& + p^2 \cdot [\text{sent} + 1 = \text{N} \wedge \text{failed} < \text{MAX}] \\
& + p^4 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 2 < \text{N} \wedge \text{failed} < \text{MAX} \wedge \text{MAX} = 1] \\
& + p^2 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 1 < \text{N} \wedge \text{failed} < \text{MAX} \wedge \text{MAX} = 1] \\
& + p^2 \cdot (1+p)^2 \cdot (1-p)^2 \cdot [\text{sent} + 1 < \text{N} \wedge \text{failed} < \text{MAX} \wedge \text{MAX} = 2] \\
& + p^4 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 2 = \text{N} \wedge \text{failed} + 1 < \text{MAX}] \\
& + p^2 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 1 = \text{N} \wedge \text{failed} + 1 < \text{MAX}] \\
& + p^2 \cdot (1+p)^2 \cdot (1-p)^2 \cdot [\text{sent} + 1 = \text{N} \wedge \text{failed} + 2 < \text{MAX}] \\
& + p^4 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 2 = \text{N} \wedge \text{failed} < \text{MAX} \wedge \text{MAX} > 1] \\
& + (1+p) \cdot (1-p) \cdot [\text{sent} < \text{N} \wedge \text{failed} + 1 = \text{MAX}] \\
& + (1+p)^2 \cdot (1-p)^2 \cdot [\text{sent} < \text{N} \wedge \text{failed} + 2 = \text{MAX}] \\
& + (1+p)^3 \cdot (1-p)^3 \cdot [\text{sent} < \text{N} \wedge \text{failed} + 3 = \text{MAX}] \\
& + [\text{sent} \geq \text{N} \vee \text{failed} \geq \text{MAX}]
\end{aligned}
$$

$$
\begin{aligned}
\Phi^5 = {} & p^8 \cdot [\text{sent} + 4 = \text{N} \wedge \text{failed} < \text{MAX}] \\
& + p^6 \cdot [\text{sent} + 3 = \text{N} \wedge \text{failed} < \text{MAX}] \\
& + p^4 \cdot [\text{sent} + 2 = \text{N} \wedge \text{failed} < \text{MAX}] \\
& + p^2 \cdot [\text{sent} + 1 = \text{N} \wedge \text{failed} < \text{MAX}] \\
& + p^6 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 3 < \text{N} \wedge \text{failed} < \text{MAX} \wedge \text{MAX} = 1] \\
& + p^4 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 2 < \text{N} \wedge \text{failed} < \text{MAX} \wedge \text{MAX} = 1] \\
& + p^2 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 1 < \text{N} \wedge \text{failed} < \text{MAX} \wedge \text{MAX} = 1] \\
& + p^2 \cdot (1+p)^3 \cdot (1-p)^3 \cdot [\text{sent} + 1 < \text{N} \wedge \text{failed} < \ \text{MAX} \wedge \text{MAX} = 3] \\
& + p^4 \cdot (1+p)^2 \cdot (1-p)^2 \cdot [\text{sent} + 2 < \text{N} \wedge \text{failed} < \text{MAX} \wedge \text{MAX} = 2] \\
& + p^2 \cdot (1+p)^2 \cdot (1-p)^2 \cdot [\text{sent} + 1 < \text{N} \wedge \text{failed} < \text{MAX} \wedge \text{MAX} = 2] \\
& + p^4 \cdot (1+p)^2 \cdot (1-p)^2 \cdot [\text{sent} + 2 < \text{N} \wedge \text{failed} + 1 < \ \text{MAX} \wedge \text{MAX} = 1] \\
& + p^2 \cdot (1+p)^2 \cdot (1-p)^2 \cdot [\text{sent} + 1 < \text{N} \wedge \text{failed} + 1 < \ \text{MAX} \wedge \text{MAX} = 1] \\
& + p^2 \cdot (1+p)^3 \cdot (1-p)^3 \cdot [\text{sent} + 1 < \text{N} \wedge \text{failed} + 1 < \ \text{MAX} \wedge \text{MAX} = 2] \\
& + p^6 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 3 = \text{N} \wedge \text{failed} + 1 < \text{MAX}] \\
& + p^4 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 2 = \text{N} \wedge \text{failed} + 1 < \text{MAX}] \\
& + p^4 \cdot (1+p)^2 \cdot (1-p)^2 \cdot [\text{sent} + 2 = \text{N} \wedge \text{failed} + 2 < \text{MAX}] \\
& + p^2 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 1 = \text{N} \wedge \text{failed} + 1 < \text{MAX}] \\
& + p^2 \cdot (1+p)^2 \cdot (1-p)^2 \cdot [\text{sent} + 1 = \text{N} \wedge \text{failed} + 2 < \text{MAX}] \\
& + p^2 \cdot (1+p)^2 \cdot (1-p)^2 \cdot [\text{sent} + 1 = \text{N} \wedge \text{failed} + 3 < \text{MAX}] \\
& + p^6 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 3 = \text{N} \wedge \text{failed} < \text{MAX} \wedge \text{MAX} > 1] \\
& + p^6 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 3 = \text{N} \wedge \text{failed} < \text{MAX} \wedge \text{MAX} > 1] \\
& + p^4 \cdot (1+p) \cdot (1-p) \cdot [\text{sent} + 2 = \text{N} \wedge \text{failed} < \text{MAX} \wedge \text{MAX} > 1] \\
& + p^4 \cdot (1+p)^2 \cdot (1-p)^2 \cdot [\text{sent} + 2 = \text{N} \wedge \text{failed} < \text{MAX} \wedge \text{MAX} > 2] \\
& \ldots \text{(continues on next page)}
\end{aligned}
$$

$$
\begin{aligned}
&\dots \text{(continued from previous page)} \\
&+ p^4 \cdot (1+p)^2 \cdot (1-p)^2 \cdot [\text{sent} + 2 = \text{N} \land \text{failed} + 1 < \text{MAX} \land \text{MAX} > 1] \\
&+ (1+p) \cdot (1-p) \cdot [\text{sent} < \text{N} \land \text{failed} + 1 = \text{MAX}] \\
&+ (1+p)^2 \cdot (1-p)^2 \cdot [\text{sent} < \text{N} \land \text{failed} + 2 = \text{MAX}] \\
&+ (1+p)^3 \cdot (1-p)^3 \cdot [\text{sent} < \text{N} \land \text{failed} + 3 = \text{MAX}] \\
&+ (1+p)^4 \cdot (1-p)^4 \cdot [\text{sent} < \text{N} \land \text{failed} + 4 = \text{MAX}] \\
&+ [\text{sent} \geq \text{N} \lor \text{failed} \geq \text{MAX}]
\end{aligned}
$$

A.2 SendPacket

This section presents the fixed-point iteration for the while-loop of $\text{wp}[\![\texttt{sendPacket}]\!](g)$, where g is the postexpectation, which differs between subsections. The pGCL program `sendPacket` is provided in Listing 4.1 on page 23. For each of the following subsections, the loop-characteristic function is the following:

$$
\begin{aligned}
\Phi_{g,\texttt{sp}}(X) &= [\text{failed} < \text{MAX} \land \neg\text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X) \\
&\quad + [\text{failed} \geq \text{MAX} \lor \text{success}] \cdot g \\
\text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X) &= p \cdot X[\text{success} := \mathit{true}] + (1-p) \cdot X[\text{failed} := \text{failed} + 1]
\end{aligned}
$$

Throughout this section, the notation $\Phi_{g,\texttt{sp}}$ is abbreviated to Φ to improve readability.

A.2.1 Probability of Success

This subsection considers the fixed-point iteration for $\text{wp}[\![\texttt{sendPacket}]\!]([\texttt{success}])$. The corresponding verification procedure is discussed in detail in Section 5.2.2. For this post-expectation, the loop-characteristic function simplifies as follows:

$$
\Phi(X) = [\text{failed} < \text{MAX} \land \neg\text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X) + [\text{success}]
$$

Fixed-point iteration

$$
\begin{aligned}
\Phi^1 &= [\text{success}] \\
\Phi^2 &= p \cdot [\text{failed} < \text{MAX} \land \neg\text{success}] + [\text{success}] \\
&\quad + p \cdot (1-p) \cdot [\text{failed} + 1 < \text{MAX} \land \neg\text{success}] \\
&\quad + [\text{success}] \\
\Phi^4 &= p \cdot [\text{failed} < \text{MAX} \land \neg\text{success}] \\
&\quad + p \cdot (1-p) \cdot [\text{failed} + 1 < \text{MAX} \land \neg\text{success}] \\
&\quad + p \cdot (1-p)^2 \cdot [\text{failed} + 2 < \text{MAX} \land \neg\text{success}] \\
&\quad + [\text{success}]
\end{aligned}
$$

Based on these equations, the expression for Φ^n is formulated as:

$$
\Phi^n = [\text{failed} < \text{MAX} \land \neg\text{success}] \cdot \sum_{i=0}^{n-2} p \cdot (1-p)^i \cdot [\text{failed} + i < \text{MAX}] + [\text{success}]
$$

A.2.2 Expected Failed Transmissions

This subsection considers the fixed-point iteration for wp⟦`sendPacket`⟧(`failed`). The corresponding verification procedure is discussed in detail in Section 5.2.3. For this postexpectation, the loop-characteristic function simplifies as follows:

$$\Phi(X) = [\text{failed} < \text{MAX} \land \neg\text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X) + [\text{failed} \geq \text{MAX} \lor \text{success}] \cdot \text{failed}$$

Fixed-point iteration

$$\begin{aligned}
\Phi^1 &= [\text{failed} \geq \text{MAX} \lor \text{success}] \cdot \text{failed} \\
\Phi^2 &= [\text{failed} < \text{MAX} \land \neg\text{success}] \cdot p \cdot \text{failed} \\
&\quad + [\text{failed} < \text{MAX} \land \text{failed} + 1 \geq \text{MAX} \land \neg\text{success}] \cdot (1-p) \cdot (\text{failed} + 1) \\
&\quad + [\text{failed} \geq \text{MAX} \lor \text{success}] \cdot \text{failed} \\
\Phi^3 &= [\text{failed} < \text{MAX} \land \neg\text{success}] \cdot p \cdot \text{failed} \\
&\quad + [\text{failed} < \text{MAX} \land \text{failed} + 1 \geq \text{MAX} \land \neg\text{success}] \cdot (1-p) \cdot (\text{failed} + 1) \\
&\quad + [\text{failed} + 1 < \text{MAX} \land \neg\text{success}] \cdot p \cdot (1-p) \cdot (\text{failed} + 1) \\
&\quad + [\text{failed} + 1 < \text{MAX} \land \text{failed} + 2 \geq \text{MAX} \land \neg\text{success}] \cdot (1-p)^2 \cdot (\text{failed} + 2) \\
&\quad + [\text{failed} \geq \text{MAX} \lor \text{success}] \cdot \text{failed} \\
\Phi^4 &= [\text{failed} < \text{MAX} \land \neg\text{success}] \cdot p \cdot \text{failed} \\
&\quad + [\text{failed} < \text{MAX} \land \text{failed} + 1 \geq \text{MAX} \land \neg\text{success}] \cdot (1-p) \cdot (\text{failed} + 1) \\
&\quad + [\text{failed} + 1 < \text{MAX} \land \neg\text{success}] \cdot p \cdot (1-p) \cdot (\text{failed} + 1) \\
&\quad + [\text{failed} + 1 < \text{MAX} \land \text{failed} + 2 \geq \text{MAX} \land \neg\text{success}] \cdot (1-p)^2 \cdot (\text{failed} + 2) \\
&\quad + [\text{failed} + 2 < \text{MAX} \land \neg\text{success}] \cdot p \cdot (1-p)^2 \cdot (\text{failed} + 2) \\
&\quad + [\text{failed} + 2 < \text{MAX} \land \text{failed} + 3 \geq \text{MAX} \land \neg\text{success}] \cdot (1-p)^3 \cdot (\text{failed} + 3) \\
&\quad + [\text{failed} \geq \text{MAX} \lor \text{success}] \cdot \text{failed}
\end{aligned}$$

Based on the above equations, the expression for Φ^n is formulated as:

$$\begin{aligned}
\Phi^n = [\text{failed} < \text{MAX} \land \neg\text{success}] \cdot \Bigg(&\sum_{i=0}^{n-2} [\text{failed} + i < \text{MAX}] \cdot p \cdot (1-p)^i \cdot (\text{failed} + i) \\
&+ \sum_{i=0}^{n-2} [\text{failed} + i + 1 = \text{MAX}] \cdot (1-p)^{i+1} \cdot (\text{failed} + i + 1) \Bigg) \\
&+ [\text{failed} \geq \text{MAX} \lor \text{success}] \cdot \text{failed}
\end{aligned}$$

A.3 BRP

This section provides the fixed-point iteration for the while-loop of $\text{wp}[\![\texttt{BRP}]\!](g)$, where g denotes the postexpectation, which differs between subsections. The pGCL program `BRP` is given in Listing 4.2 on page 24. For each of the following subsections, the loop-characteristic function is the following:

$$\begin{aligned}\Phi_{g,\texttt{brp}}(X) &= [\text{sent} < \text{N} \wedge \text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot g \\ \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) &= s \cdot X[\text{sent} := \text{sent} + 1; \text{totalFailed} := \text{totalFailed} + f; \text{success} := \mathit{true}] \\ &\quad + (1-s) \cdot X[\text{totalFailed} := \text{totalFailed} + f; \text{success} := \mathit{false}]\end{aligned}$$

Throughout this section, the notation $\Phi_{g,\texttt{brp}}$ is abbreviated to Φ to improve readability.

A.3.1 Probability of Success

This subsection considers the fixed-point iteration for $\text{wp}[\![\texttt{BRP}]\!]([\texttt{success}])$. The corresponding verification procedure is discussed in detail in Section 5.3.2. For this postexpectation, the loop-characteristic function simplifies as follows:

$$\Phi(X) = [\text{sent} < \text{N} \wedge \text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) + [\text{sent} \geq \text{N} \wedge \text{success}]$$

Fixed-point iteration

$$\begin{aligned}\Phi^1 &= [\text{sent} \geq \text{N} \wedge \text{success}] \\ \Phi^2 &= s \cdot [\text{sent} < \text{N} \wedge \text{sent} + 1 \geq \text{N} \wedge \text{success}] + [\text{sent} \geq \text{N} \wedge \text{success}] \\ \Phi^3 &= s^2 \cdot [\text{sent} + 1 < \text{N} \wedge \text{sent} + 2 \geq \text{N} \wedge \text{success}] \\ &\quad + s \cdot [\text{sent} < \text{N} \wedge \text{sent} + 1 \geq \text{N} \wedge \text{success}] \\ &\quad + [\text{sent} \geq \text{N} \wedge \text{success}] \\ \Phi^4 &= s^3 \cdot [\text{sent} + 2 < \text{N} \wedge \text{sent} + 3 \geq \text{N} \wedge \text{success}] \\ &\quad + s^2 \cdot [\text{sent} + 1 < \text{N} \wedge \text{sent} + 2 \geq \text{N} \wedge \text{success}] \\ &\quad + s \cdot [\text{sent} < \text{N} \wedge \text{sent} + 1 \geq \text{N} \wedge \text{success}] \\ &\quad + [\text{sent} \geq \text{N} \wedge \text{success}]\end{aligned}$$

Based on these equations, the expression for Φ^n is formulated and simplified as:

$$\begin{aligned}\Phi^n &= \sum_{i=0}^{n-1} s^i \cdot [\text{sent} + i - 1 < \text{N} \wedge \text{success} \wedge \text{sent} + i \geq \text{N}] + [\text{sent} \geq \text{N} \wedge \text{success}] \\ &= [\text{sent} < \text{N} \wedge \text{success}] \cdot \sum_{i=0}^{n-1} s^i \cdot [\text{sent} + i - 1 < \text{N} \wedge \text{sent} + i \geq \text{N}] + [\text{sent} \geq \text{N} \wedge \text{success}] \\ &= [\text{sent} < \text{N} \wedge \text{success}] \cdot \sum_{i=0}^{n-1} s^i \cdot [\text{sent} + i = \text{N}] + [\text{sent} \geq \text{N} \wedge \text{success}]\end{aligned}$$

A.3.2 Expected Failed Transmissions

This subsection considers the fixed-point iteration for $\mathrm{wp}[\![\texttt{BRP}]\!](\texttt{totalFailed})$. The corresponding verification procedure is discussed in detail in Section 5.3.3. For this postexpectation, the loop-characteristic function simplifies as follows:

$$\Phi(X) = [\text{sent} < \text{N} \wedge \text{success}] \cdot \mathrm{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed}$$

Fixed-point iteration

$$\begin{aligned}
\Phi^1 &= [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \\
\Phi^2 &= [\text{success} \wedge \text{sent} < \text{N} \wedge \text{sent} + 1 \geq \text{N}] \cdot s \cdot (\text{totalFailed} + f) \\
&\quad + [\text{success} \wedge \text{sent} < \text{N}] \cdot (1 - s) \cdot (\text{totalFailed} + f) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \\
\Phi^3 &= [\text{success} \wedge \text{sent} + 1 < \text{N} \wedge \text{sent} + 2 \geq \text{N}] \cdot s^2 \cdot (\text{totalFailed} + 2 \cdot f) \\
&\quad + [\text{success} \wedge \text{sent} + 1 < \text{N}] \cdot s \cdot (1 - s) \cdot (\text{totalFailed} + 2 \cdot f) \\
&\quad + [\text{success} \wedge \text{sent} < \text{N} \wedge \text{sent} + 1 \geq \text{N}] \cdot s \cdot (\text{totalFailed} + f) \\
&\quad + [\text{success} \wedge \text{sent} < \text{N}] \cdot (1 - s) \cdot (\text{totalFailed} + f) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \\
\Phi^4 &= [\text{success} \wedge \text{sent} + 2 < \text{N} \wedge \text{sent} + 3 \geq \text{N}] \cdot s^3 \cdot (\text{totalFailed} + 3 \cdot f) \\
&\quad + [\text{success} \wedge \text{sent} + 2 < \text{N}] \cdot s^2 \cdot (1 - s) \cdot (\text{totalFailed} + 3 \cdot f) \\
&\quad + [\text{success} \wedge \text{sent} + 1 < \text{N} \wedge \text{sent} + 2 \geq \text{N}] \cdot s^2 \cdot (\text{totalFailed} + 2 \cdot f) \\
&\quad + [\text{success} \wedge \text{sent} + 1 < \text{N}] \cdot s \cdot (1 - s) \cdot (\text{totalFailed} + 2 \cdot f) \\
&\quad + [\text{success} \wedge \text{sent} < \text{N} \wedge \text{sent} + 1 \geq \text{N}] \cdot s \cdot (\text{totalFailed} + f) \\
&\quad + [\text{success} \wedge \text{sent} < \text{N}] \cdot (1 - s) \cdot (\text{totalFailed} + f) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed}
\end{aligned}$$

Based on these equations, the expression for Φ^n is formulated and simplified as follows. The changes made in each simplification step are colour-coded for easy reference.

$$\begin{aligned}
\Phi^n &= [\text{success}] \cdot \left(\sum_{i=0}^{n-2} [\text{sent} + i < \text{N}] \cdot s^i \cdot (1-s) \cdot (\text{totalFailed} + (i+1) \cdot f) \right. \\
&\qquad \left. + \sum_{i=0}^{n-2} [\text{sent} + i < \text{N} \wedge \text{sent} + i + 1 \geq \text{N}] \cdot s^{i+1} \cdot (\text{totalFailed} + (i+1) \cdot f) \right) \\
&\quad + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{totalFailed} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \left(\sum_{i=0}^{n-2} [\text{sent} + i < \text{N}] \cdot s^i \cdot (1-s) \cdot (\text{totalFailed} + (i+1) \cdot f) \right. \\
&\qquad \left. + \sum_{i=0}^{n-2} [\text{sent} + i < \text{N} \wedge \text{sent} + i + 1 \geq \text{N}] \cdot s^{i+1} \cdot (\text{totalFailed} + (i+1) \cdot f) \right) \\
&\quad + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{totalFailed} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \left(\sum_{i=0}^{n-2} [\text{sent} + i < \text{N}] \cdot s^i \cdot (1-s) \cdot (\text{totalFailed} + (i+1) \cdot f) \right. \\
&\qquad \left. + \sum_{i=0}^{n-2} [\text{sent} + i + 1 = \text{N}] \cdot s^{i+1} \cdot (\text{totalFailed} + (i+1) \cdot f) \right) \\
&\quad + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{totalFailed}
\end{aligned}$$

A.3.3 Expected Sent Packets

This subsection considers the fixed-point iteration for $\text{wp}[\![\texttt{BRP}]\!](\texttt{sent})$. The corresponding verification procedure is discussed in detail in Section 5.3.4. For this postexpectation, the loop-characteristic function simplifies as follows:

$$\Phi(X) = [\text{sent} < \text{N} \wedge \text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{sent}$$

Fixed-point iteration

$$\begin{aligned}
\Phi^1 &= [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{sent} \\
\Phi^2 &= [\text{success} \wedge \text{sent} < \text{N} \wedge \text{sent} + 1 \geq \text{N}] \cdot s \cdot (\text{sent} + 1) \\
&\quad + [\text{success} \wedge \text{sent} < \text{N}] \cdot (1-s) \cdot \text{sent} \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{sent} \\
\Phi^3 &= [\text{success} \wedge \text{sent} + 1 < \text{N} \wedge \text{sent} + 2 \geq \text{N}] \cdot s^2 \cdot (\text{sent} + 2) \\
&\quad + [\text{success} \wedge \text{sent} + 1 < \text{N}] \cdot s \cdot (1-s) \cdot (\text{sent} + 1) \\
&\quad + [\text{success} \wedge \text{sent} < \text{N} \wedge \text{sent} + 1 \geq \text{N}] \cdot s \cdot (\text{sent} + 1) \\
&\quad + [\text{success} \wedge \text{sent} < \text{N}] \cdot (1-s) \cdot \text{sent} \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{sent}
\end{aligned}$$

$$
\begin{aligned}
\Phi^4 = {} & [\text{success} \wedge \text{sent} + 2 < \text{N} \wedge \text{sent} + 3 \geq \text{N}] \cdot s^3 \cdot (\text{sent} + 3) \\
& + [\text{success} \wedge \text{sent} + 2 < \text{N}] \cdot s^2 \cdot (1 - s) \cdot (\text{sent} + 2) \\
& + [\text{success} \wedge \text{sent} + 1 < \text{N} \wedge \text{sent} + 2 \geq \text{N}] \cdot s^2 \cdot (\text{sent} + 2) \\
& + [\text{success} \wedge \text{sent} + 1 < \text{N}] \cdot s \cdot (1 - s) \cdot (\text{sent} + 1) \\
& + [\text{success} \wedge \text{sent} < \text{N} \wedge \text{sent} + 1 \geq \text{N}] \cdot s \cdot (\text{sent} + 1) \\
& + [\text{success} \wedge \text{sent} < \text{N}] \cdot (1 - s) \cdot \text{sent} \\
& + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{sent}
\end{aligned}
$$

Based on these equations, the expression for Φ^n is derived and subsequently simplified in a step-by-step manner. For clarity, each simplification step is colour-coded to highlight the modifications.

$$
\begin{aligned}
\Phi^n = {} & [\text{success}] \cdot \Big(\sum_{i=0}^{n-2} [\text{sent} + i < \text{N}] \cdot s^i \cdot (1 - s) \cdot (\text{sent} + i) \\
& \quad + \sum_{i=0}^{n-2} [\text{sent} + i < \text{N} \wedge \text{sent} + i + 1 \geq \text{N}] \cdot s^{i+1} \cdot (\text{sent} + i + 1) \Big) \\
& + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent} \\
= {} & [\text{success} \wedge \text{sent} < \text{N}] \cdot \Big(\sum_{i=0}^{n-2} [\text{sent} + i < \text{N}] \cdot s^i \cdot (1 - s) \cdot (\text{sent} + i) \\
& \quad + \sum_{i=0}^{n-2} [\text{sent} + i < \text{N} \wedge \text{sent} + i + 1 \geq \text{N}] \cdot s^{i+1} \cdot (\text{sent} + i + 1) \Big) \\
& + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent} \\
= {} & [\text{success} \wedge \text{sent} < \text{N}] \cdot \Big(\sum_{i=0}^{n-2} [\text{sent} + i < \text{N}] \cdot s^i \cdot (1 - s) \cdot (\text{sent} + i) \\
& \quad + \sum_{i=0}^{n-2} [\text{sent} + i + 1 = \text{N}] \cdot s^{i+1} \cdot (\text{sent} + i + 1) \Big) \\
& + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent}
\end{aligned}
$$

APPENDIX B

SUPREMUM SIMPLIFICATION

This chapter presents the step-by-step simplification of supremum expressions arising from the while-loops in the following weakest preexpectation computations: wp⟦`sendPacket`⟧(failed), wp⟦`BRP`⟧(totalFailed), and wp⟦`BRP`⟧(sent). Each step is accompanied by a brief explanation. Whenever expressions are particularly long or changes are not immediately clear, the relevant parts are highlighted for clarity.

Throughout this chapter, the geometric sum Equation 5.2 is applied repeatedly and is therefore included below for reference:

$$\sum_{k=0}^{n}(1-r)\cdot b\cdot r^k = b\cdot(1-r^{n+1}) \tag{B.1}$$

Its equivalence to the well-known geometric sum function (Equation 7.4) is established on page 35. A second geometric sum, also used in the following sections, is given below:

$$\sum_{i=0}^{n-1} i\cdot r^i = \frac{r-n\cdot r^n+(n-1)\cdot r^{n+1}}{(1-r)^2} \text{ if } |r|<1 \tag{B.2}$$

B.1 SendPacket Failed

This section provides the detailed simplification of the supremum of the while-loop of wp⟦`sendPacket`⟧(failed), which is omitted in Section 5.2.3.

$$\begin{aligned}
&\sup_{n\in\mathbb{N}}\Phi^n \\
&= [\text{failed}<\text{MAX}\wedge\neg\text{success}]\cdot\left(\sum_{i=0}^{\infty}[\text{failed}+i<\text{MAX}]\cdot p\cdot(1-p)^i\cdot(\text{failed}+i)\right. \\
&\quad\left.+\sum_{i=0}^{\infty}[\text{failed}+i+1=\text{MAX}]\cdot(1-p)^{i+1}\cdot(\text{failed}+i+1)\right) \\
&\quad+[\text{failed}\geq\text{MAX}\vee\text{success}]\cdot\text{failed} \\
&= \underbrace{[\text{failed}<\text{MAX}\wedge\neg\text{success}]\cdot\sum_{i=0}^{\infty}[\text{failed}+i<\text{MAX}]\cdot p\cdot(1-p)^i\cdot(\text{failed}+i)}_{S_1} \\
&\quad+\underbrace{[\text{failed}<\text{MAX}\wedge\neg\text{success}]\cdot\sum_{i=0}^{\infty}[\text{failed}+i+1=\text{MAX}]\cdot(1-p)^{i+1}\cdot(\text{failed}+i+1)}_{S_2} \\
&\quad+[\text{failed}\geq\text{MAX}\vee\text{success}]\cdot\text{failed}
\end{aligned}$$

Due to the complexity of the expression, the supremum is divided into two parts, denoted S_1 and S_2, which are simplified independently before being substituted back into the main equation.

$$
\begin{aligned}
S_1 &= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \sum_{i=0}^{\infty} [\text{failed} + i < \text{MAX}] \cdot p \cdot (1-p)^i \cdot (\text{failed} + i) \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \sum_{i=0}^{\text{MAX}-\text{failed}-1} p \cdot (1-p)^i \cdot (\text{failed} + i) \quad \text{(Restrict the sum)} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \Bigg(\underbrace{\sum_{i=0}^{\text{MAX}-\text{failed}-1} p \cdot \text{failed} \cdot (1-p)^i}_{S_{1a}} + \underbrace{\sum_{i=0}^{\text{MAX}-\text{failed}-1} p \cdot i \cdot (1-p)^i}_{S_{1b}} \Bigg) \quad \text{(Expand the sum)}
\end{aligned}
$$

To restrict the sum, it must be ensured that the upper bound remains greater than or equal to the lower bound, i.e. $\text{MAX} - \text{failed} - 1 \geq 0$. This condition is satisfied by analysing the guard preceding the summation:

$$
\begin{aligned}
\text{failed} < \text{MAX} &\iff 0 < \text{MAX} - \text{failed} \\
&\iff 0 \leq \text{MAX} - \text{failed} - 1
\end{aligned}
$$

As shown, this inequality holds under the given guard in the equation for S_1, justifying the restriction of the summation.

The term S_1 is further decomposed into S_{1a} and S_{1b} which are simplified individually and then recombined to yield the simplified form of S_1.

The summation S_{1a} can be simplified using Equation B.1. In contrast, the summation S_{1b} includes an additional factor of i, necessitating the use of Equation B.2. For this equation to be applicable, the absolute value of the common ratio must be strictly less than 1, i.e. $|r| < 1$.

In the case of S_{1b}, the common ratio r is given by $(1-p)$, which represents a probability. Hence, it satisfies $0 \leq (1-p) \leq 1$. To ensure that $|r| < 1$, it suffices to assume that $(1-p) \neq 1$, i.e. $p > 0$. This assumption is also made and justified in Section 5.2.3.

Under this assumption, the summations S_{1a}, and S_{1b} can be evaluated using the appropriate geometric sum formulas, which the condition $p > 0$ explicitly applied in the simplification of S_{1b}.

$$
\begin{aligned}
S_{1a} &= \sum_{i=0}^{\text{MAX}-\text{failed}-1} p \cdot \text{failed} \cdot (1-p)^i \\
&= \text{failed} \cdot (1 - (1-p)^{\text{MAX}-\text{failed}}) && \text{(Apply B.1)} \\
&= \text{failed} - \text{failed} \cdot (1-p)^{\text{MAX}-\text{failed}} && \text{(Distribute)}
\end{aligned}
$$

$$
\begin{aligned}
S_{1b} &= \sum_{i=0}^{\text{MAX}-\text{failed}-1} p \cdot i \cdot (1-p)^i \\
&= p \cdot \sum_{i=0}^{\text{MAX}-\text{failed}-1} i \cdot (1-p)^i && \text{(Factor out } p\text{)} \\
&= p \cdot \frac{1}{p^2} \cdot \Big((1-p) - (\text{MAX} - \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}} \\
&\quad + (\text{MAX} - \text{failed} - 1) \cdot (1-p)^{\text{MAX}-\text{failed}+1} \Big) && \text{(Apply B.2, } p > 0\text{)} \\
&= \frac{1}{p} \cdot \Big((1-p) - (\text{MAX} - \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}} \\
&\quad + (\text{MAX} - \text{failed} - 1) \cdot (1-p)^{\text{MAX}-\text{failed}+1} \Big) && \text{(Simplify)}
\end{aligned}
$$

$$
\begin{aligned}
S_1 &= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot (S_{1a} + S_{1b}) \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \Bigg(\text{failed} - \text{failed} \cdot (1-p)^{\text{MAX}-\text{failed}} \\
&\quad + \frac{1}{p} \cdot \Big((1-p) - (\text{MAX} - \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}} \\
&\quad + (\text{MAX} - \text{failed} - 1) \cdot (1-p)^{\text{MAX}-\text{failed}+1} \Big) \Bigg) && \text{(Substitute)} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \Bigg(\frac{p}{p} \cdot (\text{failed} - \text{failed} \cdot (1-p)^{\text{MAX}-\text{failed}}) \\
&\quad + \frac{1}{p} \cdot \Big((1-p) - (\text{MAX} - \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}} \\
&\quad + (\text{MAX} - \text{failed} - 1) \cdot (1-p)^{\text{MAX}-\text{failed}+1} \Big) \Bigg) && \text{(Multiply by 1)} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \frac{1}{p} \cdot \Big(p \cdot \text{failed} - p \cdot \text{failed} \cdot (1-p)^{\text{MAX}-\text{failed}} \\
&\quad + (1-p) - (\text{MAX} - \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}} \\
&\quad + (\text{MAX} - \text{failed} - 1) \cdot (1-p)^{\text{MAX}-\text{failed}+1} \Big) && \text{(Distribute)}
\end{aligned}
$$

After completing the simplification of S_1, attention shifts to S_2, which is treated analogously:

$$
\begin{aligned}
S_2 &= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \sum_{i=0}^{\infty} [\text{failed} + i + 1 = \text{MAX}] \cdot (1-p)^{i+1} \\
&\quad \cdot (\text{failed} + i + 1) \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot (1-p)^{\text{MAX}-\text{failed}-1+1} \\
&\quad \cdot (\text{failed} + \text{MAX} - \text{failed} - 1 + 1) && \text{(Restrict sum)} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot (1-p)^{\text{MAX}-\text{failed}} \cdot \text{MAX} && \text{(Simplify)}
\end{aligned}
$$

Finally, having simplified both S_1 and S_2, their results are combined to complete the simplification of the original supremum expression.

$$
\begin{aligned}
S_1 + S_2 = {} & [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \frac{1}{p} \cdot \Bigg(p \cdot \text{failed} - p \cdot \text{failed} \cdot (1-p)^{\text{MAX}-\text{failed}} \\
& + (1-p) - (\text{MAX} - \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}} \\
& + (\text{MAX} - \text{failed} - 1) \cdot (1-p)^{\text{MAX}-\text{failed}+1} \Bigg) \\
& + [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot (1-p)^{\text{MAX}-\text{failed}} \cdot \text{MAX} && \text{(Substitute)} \\
= {} & [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \frac{1}{p} \cdot \Bigg(p \cdot \text{failed} - p \cdot \text{failed} \cdot (1-p)^{\text{MAX}-\text{failed}} \\
& + (1-p) - (\text{MAX} - \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}} \\
& + (\text{MAX} - \text{failed} - 1) \cdot (1-p)^{\text{MAX}-\text{failed}+1} \Bigg) \\
& + [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \frac{p}{p} \cdot (1-p)^{\text{MAX}-\text{failed}} \cdot \text{MAX} && \text{(Multiply } S_2 \text{ by 1)} \\
= {} & [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \frac{1}{p} \cdot \Bigg(p \cdot \text{failed} - p \cdot \text{failed} \cdot (1-p)^{\text{MAX}-\text{failed}} \\
& + (1-p) - (\text{MAX} - \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}} \\
& + (\text{MAX} - \text{failed} - 1) \cdot (1-p)^{\text{MAX}-\text{failed}+1} \\
& + p \cdot (1-p)^{\text{MAX}-\text{failed}} \cdot \text{MAX} \Bigg) && \text{(Distribute)} \\
= {} & [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \frac{1}{p} \cdot \Bigg(p \cdot \text{failed} + (1-p) \\
& - (\text{MAX} - \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}} \\
& + (\text{MAX} - \text{failed} - 1) \cdot (1-p)^{\text{MAX}-\text{failed}+1} \\
& + p \cdot \text{MAX} \cdot (1-p)^{\text{MAX}-\text{failed}} - p \cdot \text{failed} \cdot (1-p)^{\text{MAX}-\text{failed}} \Bigg) && \text{(Reorder)} \\
= {} & [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \frac{1}{p} \cdot \Bigg(p \cdot \text{failed} + (1-p) \\
& - (\text{MAX} - \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}} \\
& + (\text{MAX} - \text{failed} - 1) \cdot (1-p)^{\text{MAX}-\text{failed}+1} \\
& + p \cdot (\text{MAX} - \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}} \Bigg) && \text{(Distribute)}
\end{aligned}
$$

... (continues on next page)

... (continued from previous page)

$$
\begin{aligned}
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \frac{1}{p} \cdot \Big(p \cdot \text{failed} + (1-p) \\
&\quad + (\text{MAX} - \text{failed} - 1) \cdot (1-p)^{\text{MAX}-\text{failed}+1} \\
&\quad - (\text{MAX} - \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}} \\
&\quad + p \cdot (\text{MAX} - \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}} \Big) && \text{(Reorder)} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \frac{1}{p} \cdot \Big(p \cdot \text{failed} + (1-p) \\
&\quad + (\text{MAX} - \text{failed} - 1) \cdot (1-p)^{\text{MAX}-\text{failed}+1} \\
&\quad - (1-p) \cdot (\text{MAX} - \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}} \Big) && \text{(Distribute)} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \frac{1}{p} \cdot \Big(p \cdot \text{failed} + (1-p) \\
&\quad + (\text{MAX} - \text{failed} - 1) \cdot (1-p)^{\text{MAX}-\text{failed}+1} \\
&\quad - (\text{MAX} - \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}+1} \Big) && \text{(Simplify)} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \frac{1}{p} \cdot \Big(p \cdot \text{failed} + (1-p) \\
&\quad + (\text{MAX} - \text{failed} - 1 - \text{MAX} + \text{failed}) \cdot (1-p)^{\text{MAX}-\text{failed}+1} \Big) && \text{(Distribute)} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \frac{1}{p} \cdot \Big(p \cdot \text{failed} + (1-p) \\
&\quad - (1-p)^{\text{MAX}-\text{failed}+1} \Big) && \text{(Simplify)} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \left(\text{failed} + \frac{(1-p) - (1-p)^{\text{MAX}-\text{failed}+1}}{p} \right) && \text{(Simplify)}
\end{aligned}
$$

We now revisit the assumption made during the application of Equation B.2, which was introduced to avoid division by 0. However, upon examining the simplified expression, it becomes evident that when $p = 0$, the numerator of the fraction also evaluates to zero. As a result, the assumption $p > 0$ is no longer required.

We thus arrive at the simplified from of the supremum of wp$[\![$`sendPacket`$]\!]$(failed):

$$
\begin{aligned}
\sup_{n \in \mathbb{N}} \Phi^n &= (S_1 + S_2) + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \left(\text{failed} + \frac{(1-p) - (1-p)^{\text{MAX}-\text{failed}+1}}{p} \right) \\
&\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed}
\end{aligned}
$$

B.2 BRP TotalFailed

This section provides the detailed simplification of the supremum of the while-loop of wp⟦BRP⟧(totalFailed), which is omitted in Section 5.3.3.

$$
\begin{aligned}
&\sup_{n\in\mathbb{N}}\Phi^n \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \left(\sum_{i=0}^{\infty} [\text{sent} + i < \text{N}] \cdot s^i \cdot (1-s) \cdot (\text{totalFailed} + (i+1)\cdot f) \right. \\
&\qquad \left. + \sum_{i=0}^{\infty} [\text{sent} + i + 1 = \text{N}] \cdot s^{i+1} \cdot (\text{totalFailed} + (i+1)\cdot f) \right) \\
&\quad + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{totalFailed} \\
&= \underbrace{[\text{success} \wedge \text{sent} < \text{N}] \cdot \sum_{i=0}^{\infty} [\text{sent} + i < \text{N}] \cdot s^i \cdot (1-s) \cdot (\text{totalFailed} + (i+1)\cdot f)}_{S_1} \\
&\quad + \underbrace{[\text{success} \wedge \text{sent} < \text{N}] \cdot \sum_{i=0}^{\infty} [\text{sent} + i + 1 = \text{N}] \cdot s^{i+1} \cdot (\text{totalFailed} + (i+1)\cdot f)}_{S_2} \\
&\quad + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{totalFailed}
\end{aligned}
$$

Due to the complexity of the expression, the supremum is divided into two parts, denoted S_1 and S_2, which are simplified independently before being substituted back into the main equation.

$$
\begin{aligned}
S_1 &= [\text{success} \wedge \text{sent} < \text{N}] \cdot \sum_{i=0}^{\infty} [\text{sent} + i < \text{N}] \cdot s^i \cdot (1-s) \cdot (\text{totalFailed} + (i+1)\cdot f) \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \sum_{i=0}^{\text{N}-\text{sent}-1} s^i \cdot (1-s) \cdot (\text{totalFailed} + (i+1)\cdot f) \\
&\qquad\qquad \text{(Restrict the sum)} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \left(\sum_{i=0}^{\text{N}-\text{sent}-1} s^i \cdot (1-s) \cdot \text{totalFailed} \right. \\
&\quad \left. + \sum_{i=0}^{\text{N}-\text{sent}-1} s^i \cdot (1-s) \cdot (i+1) \cdot f \right) \qquad \text{(Expand the sum)} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \left(\sum_{i=0}^{\text{N}-\text{sent}-1} s^i \cdot (1-s) \cdot \text{totalFailed} \right. \\
&\quad \left. + \sum_{i=0}^{\text{N}-\text{sent}-1} s^i \cdot (1-s) \cdot i \cdot f + \sum_{i=0}^{\text{N}-\text{sent}-1} s^i \cdot (1-s) \cdot 1 \cdot f \right) \qquad \text{(Expand the sum)}
\end{aligned}
$$

... (continues on next page)

... (continued from previous page)

$$= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\underbrace{\sum_{i=0}^{\text{N}-\text{sent}-1} (1-s) \cdot \text{totalFailed} \cdot s^i}_{S_{1a}} + (1-s) \cdot f \cdot \underbrace{\sum_{i=0}^{\text{N}-\text{sent}-1} i \cdot s^i}_{S_{1b}} + \underbrace{\sum_{i=0}^{\text{N}-\text{sent}-1} (1-s) \cdot f \cdot s^i}_{S_{1c}} \Bigg) \qquad \text{(Reorder)}$$

To restrict the sum, it must be ensured that the upper bound remains greater than or equal to the lower bound, i.e. $\text{N} - \text{sent} - 1 \geq 0$. This condition is satisfied by analysing the guard preceding the summation:

$$\begin{aligned} \text{sent} < \text{N} &\iff 0 < \text{N} - \text{sent} \\ &\iff 0 \leq \text{N} - \text{sent} - 1 \end{aligned}$$

As shown, this inequality holds under the given guard in the equation for S_1, justifying the restriction of the summation.

The term S_1 is further decomposed into three components, S_{1a}, S_{1b}, and S_{1c}, each of which is simplified individually before being recombined to obtain the simplified form of S_1.

The summations S_{1a} and S_{1c} can be simplified using Equation B.1. In contrast, the summation S_{1b} includes an additional factor of i, necessitating the use of Equation B.2. For this equation to be applicable, the absolute value of the common ratio must be strictly less than 1, i.e. $|r| < 1$.

In the case of S_{1b}, the common ratio r is given by s, which represents a probability. Hence, it satisfies $0 \leq s \leq 1$. To ensure that $|r| < 1$, it suffices to assume that $s \neq 1$, i.e. $s < 1$. This assumption is also made and justified in Section 5.3.3.

Under this assumption, the summations S_{1a}, S_{1b}, and S_{1c} can all be evaluated using the appropriate geometric sum formulas, with the condition $s < 1$ explicitly applied in the simplification of S_{1b}.

$$S_{1a} = \text{totalFailed} \cdot (1 - s^{\text{N}-\text{sent}}) \qquad \text{(Apply B.1)}$$

$$S_{1b} = \frac{s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1}}{(1-s)^2} \qquad \text{(Apply B.2, } s < 1\text{)}$$

$$S_{1c} = f \cdot (1 - s^{\text{N}-\text{sent}}) \qquad \text{(Apply B.1)}$$

$$\begin{aligned} S_1 &= [\text{success} \wedge \text{sent} < \text{N}] \cdot (S_{1a} + (1-s) \cdot f \cdot S_{1b} + S_{1c}) \\ &= [\text{success} \wedge \text{sent} < \text{N}] \cdot (S_{1a} + S_{1c} + (1-s) \cdot f \cdot S_{1b}) \qquad \text{(Reorder)} \\ &= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\text{totalFailed} \cdot (1 - s^{\text{N}-\text{sent}}) + f \cdot (1 - s^{\text{N}-\text{sent}}) \\ &\quad + (1-s) \cdot f \cdot \frac{s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1}}{(1-s)^2} \Bigg) \qquad \text{(Substitute)} \end{aligned}$$

After completing the simplification of S_1, attention shifts to S_2, which is treated analogously:

$$
\begin{aligned}
S_2 &= [\text{success} \wedge \text{sent} < \text{N}] \cdot \sum_{i=0}^{\infty} [\text{sent} + i + 1 = \text{N}] \cdot s^{i+1} \cdot (\text{totalFailed} + (i+1) \cdot f) \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot s^{\text{N}-\text{sent}-1+1} \cdot (\text{totalFailed} + (\text{N} - \text{sent} - 1 + 1) \cdot f) \\
&\qquad\qquad \text{(Restrict the sum)} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot s^{\text{N}-\text{sent}} \cdot (\text{totalFailed} + (\text{N} - \text{sent}) \cdot f) \qquad \text{(Simplify)}
\end{aligned}
$$

Finally, having simplified both S_1 and S_2, their results are combined to complete the simplification of the original supremum expression.

$$
\begin{aligned}
S_1 + S_2 &= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\text{totalFailed} \cdot (1 - s^{\text{N}-\text{sent}}) + f \cdot (1 - s^{\text{N}-\text{sent}}) \\
&\qquad + (1-s) \cdot f \cdot \frac{s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1}}{(1-s)^2} \Bigg) \\
&\quad + [\text{success} \wedge \text{sent} < \text{N}] \cdot s^{\text{N}-\text{sent}} \cdot (\text{totalFailed} + (\text{N} - \text{sent}) \cdot f) \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\text{totalFailed} \cdot (1 - s^{\text{N}-\text{sent}}) + f \cdot (1 - s^{\text{N}-\text{sent}}) \\
&\qquad + f \cdot \frac{s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1}}{1-s} \Bigg) \\
&\quad + [\text{success} \wedge \text{sent} < \text{N}] \cdot s^{\text{N}-\text{sent}} \cdot (\text{totalFailed} + (\text{N} - \text{sent}) \cdot f) \qquad \text{(Simplify)} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\text{totalFailed} \cdot (1 - s^{\text{N}-\text{sent}}) + f \cdot (1 - s^{\text{N}-\text{sent}}) \\
&\quad + f \cdot \frac{s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1}}{1-s} \\
&\quad + s^{\text{N}-\text{sent}} \cdot (\text{totalFailed} + (\text{N} - \text{sent}) \cdot f) \Bigg) \qquad \text{(Distribute)} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\text{totalFailed} - \text{totalFailed} \cdot s^{\text{N}-\text{sent}} + f - f \cdot s^{\text{N}-\text{sent}} \\
&\quad + f \cdot \frac{s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1}}{1-s} \\
&\quad + \text{totalFailed} \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent}) \cdot f \cdot s^{\text{N}-\text{sent}} \Bigg) \qquad \text{(Distribute)} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\text{totalFailed} - \text{totalFailed} \cdot s^{\text{N}-\text{sent}} + \text{totalFailed} \cdot s^{\text{N}-\text{sent}} \\
&\quad + f - f \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent}) \cdot f \cdot s^{\text{N}-\text{sent}} \\
&\quad + f \cdot \frac{s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1}}{1-s} \Bigg) \qquad \text{(Reorder)}
\end{aligned}
$$

... (continues on next page)

... (continued from previous page)

$$
\begin{aligned}
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\text{totalFailed} \\
&\quad + f - f \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent}) \cdot f \cdot s^{\text{N}-\text{sent}} \\
&\quad + f \cdot \frac{s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1}}{1 - s} \Bigg) && \text{(Simplify)} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\text{totalFailed} + f \cdot \Big(1 - s^{\text{N}-\text{sent}} + (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} \\
&\quad + \frac{s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1}}{1 - s} \Big) \Bigg) && \text{(Distribute)} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\text{totalFailed} + f \cdot \Big(\frac{1 - s}{1 - s} \cdot \big(1 - s^{\text{N}-\text{sent}} + (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} \big) \\
&\quad + \frac{s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1}}{1 - s} \Big) \Bigg) && \text{(Multiply by 1)} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \left(\text{totalFailed} + f \cdot \frac{A}{1 - s} \right) && \text{(Distribute)}
\end{aligned}
$$

Where $A = (1 - s) \cdot \left(1 - s^{\text{N}-\text{sent}} + (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} \right)$

$$
\begin{aligned}
&\quad + s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} \\
&= (1 - s) - (1 - s) \cdot s^{\text{N}-\text{sent}} + (1 - s) \cdot (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} \\
&\quad + s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} && \text{(Distribute)} \\
&= 1 - s + s - (1 - s) \cdot s^{\text{N}-\text{sent}} + (1 - s) \cdot (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} \\
&\quad - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} && \text{(Reorder)} \\
&= 1 + (-(1 - s) + (1 - s) \cdot (\text{N} - \text{sent}) - (\text{N} - \text{sent})) \cdot s^{\text{N}-\text{sent}} \\
&\quad + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} && \text{(Simplify, Distribute)} \\
&= 1 + (-1 + s + (\text{N} - \text{sent}) - s \cdot (\text{N} - \text{sent}) - (\text{N} - \text{sent})) \cdot s^{\text{N}-\text{sent}} \\
&\quad + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} && \text{(Distribute)} \\
&= 1 + (-1 + s - s \cdot (\text{N} - \text{sent})) \cdot s^{\text{N}-\text{sent}} \\
&\quad + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} && \text{(Simplify)} \\
&= 1 - s^{\text{N}-\text{sent}} + s \cdot s^{\text{N}-\text{sent}} - s \cdot (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} \\
&\quad + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} && \text{(Distribute)} \\
&= 1 - s^{\text{N}-\text{sent}} + s^{\text{N}-\text{sent}+1} - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}+1} \\
&\quad + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} && \text{(Simplify)} \\
&= 1 - s^{\text{N}-\text{sent}} + (1 - (\text{N} - \text{sent}) + (\text{N} - \text{sent} - 1)) \cdot s^{\text{N}-\text{sent}+1} && \text{(Distribute)} \\
&= 1 - s^{\text{N}-\text{sent}} + (1 - \text{N} + \text{sent} + \text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} && \text{(Distribute)} \\
&= 1 - s^{\text{N}-\text{sent}} && \text{(Simplify)}
\end{aligned}
$$

We thus arrive at the simplified from of the supremum of wp⟦BRP⟧(totalFailed):

$$\begin{aligned}\sup_{n\in\mathbb{N}}\Phi^n &= (S_1 + S_2) + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{totalFailed} \\ &= [\text{success} \wedge \text{sent} < \text{N}] \cdot \left(\text{totalFailed} + f \cdot \frac{1 - s^{\text{N}-\text{sent}}}{1-s}\right) \\ &\quad + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{totalFailed}\end{aligned}$$

We now revisit the assumption made during the application of Equation B.2, which was introduced to avoid division by 0. However, upon examining the simplified expression, it becomes evident that when $s = 1$, the numerator of the fraction also evaluates to zero. As a result, the assumption $s < 1$ is no longer required.

B.3 BRP Sent

This section provides the detailed simplification of the supremum of the while-loop of wp⟦BRP⟧(sent), which is omitted in Section 5.3.4.

$$\begin{aligned}\sup_{n\in\mathbb{N}}\Phi^n &= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\sum_{i=0}^{\infty} [\text{sent} + i < \text{N}] \cdot s^i \cdot (1-s) \cdot (\text{sent} + i) \\ &\qquad + \sum_{i=0}^{\infty} [\text{sent} + i + 1 = \text{N}] \cdot s^{i+1} \cdot (\text{sent} + i + 1) \Bigg) \\ &\quad + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent} \\ &= \underbrace{[\text{success} \wedge \text{sent} < \text{N}] \cdot \sum_{i=0}^{\infty} [\text{sent} + i < \text{N}] \cdot s^i \cdot (1-s) \cdot (\text{sent} + i)}_{S_1} \\ &\quad + \underbrace{[\text{success} \wedge \text{sent} < \text{N}] \cdot \sum_{i=0}^{\infty} [\text{sent} + i + 1 = \text{N}] \cdot s^{i+1} \cdot (\text{sent} + i + 1)}_{S_2} \\ &\quad + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent}\end{aligned}$$

Due to the complexity of the expression, the supremum is divided into two parts, denoted S_1 and S_2, which are simplified independently before being substituted back into the main equation.

$$\begin{aligned}S_1 &= [\text{success} \wedge \text{sent} < \text{N}] \cdot \sum_{i=0}^{\infty} [\text{sent} + i < \text{N}] \cdot s^i \cdot (1-s) \cdot (\text{sent} + i) \\ &= [\text{success} \wedge \text{sent} < \text{N}] \cdot \sum_{i=0}^{\text{N}-\text{sent}-1} s^i \cdot (1-s) \cdot (\text{sent} + i) && \text{(Restrict the sum)} \\ &= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\sum_{i=0}^{\text{N}-\text{sent}-1} s^i \cdot (1-s) \cdot \text{sent} + \sum_{i=0}^{\text{N}-\text{sent}-1} s^i \cdot (1-s) \cdot i \Bigg) \\ & && \text{(Expand the sum)} \\ &= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\underbrace{\sum_{i=0}^{\text{N}-\text{sent}-1} (1-s) \cdot \text{sent} \cdot s^i}_{S_{1a}} + (1-s) \cdot \underbrace{\sum_{i=0}^{\text{N}-\text{sent}-1} i \cdot s^i}_{S_{1b}} \Bigg) && \text{(Reorder)}\end{aligned}$$

To restrict the sum, it must be ensured that the upper bound remains greater than or equal to the lower bound, i.e. $\mathrm{N} - \mathrm{sent} - 1 \geq 0$. This condition is satisfied by analysing the guard preceding the summation:

$$\begin{aligned} \mathrm{sent} < \mathrm{N} &\iff 0 < \mathrm{N} - \mathrm{sent} \\ &\iff 0 \leq \mathrm{N} - \mathrm{sent} - 1 \end{aligned}$$

As shown, this inequality holds under the given guard in the equation for S_1, justifying the restriction of the summation.

The term S_1 is further decomposed into S_{1a} and S_{1b}, which are simplified individually before being recombined to obtain the simplified form of S_1.

The summation S_{1a} can be simplified using Equation B.1. In contrast, S_{1b} includes an additional factor of i, necessitating the use of Equation B.2. For this equation to be applicable, the absolute value of the common ratio must be strictly less than 1, i.e. $|r| < 1$.

In the case of S_{1b}, the common ratio r is given by s, which represents a probability. Hence, it satisfies $0 \leq s \leq 1$. To ensure that $|r| < 1$, it suffices to assume that $s \neq 1$, i.e. $s < 1$. This assumption is also made and justified in Section 5.3.3.

Under this assumption, the summations S_{1a} and S_{1b} can be evaluated using the appropriate geometric sum formulas, with the condition $s < 1$ explicitly applied in the simplification of S_{1b}.

$$S_{1a} = \mathrm{sent} \cdot (1 - s^{\mathrm{N}-\mathrm{sent}}) \qquad \text{(Apply B.1)}$$

$$S_{1b} = \frac{s - (\mathrm{N} - \mathrm{sent}) \cdot s^{\mathrm{N}-\mathrm{sent}} + (\mathrm{N} - \mathrm{sent} - 1) \cdot s^{\mathrm{N}-\mathrm{sent}+1}}{(1-s)^2} \qquad \text{(Apply B.2, } s < 1\text{)}$$

$$\begin{aligned} S_1 &= [\mathrm{success} \wedge \mathrm{sent} < \mathrm{N}] \cdot \Big(S_{1a} + (1-s) \cdot S_{1b} \Big) \\ &= [\mathrm{success} \wedge \mathrm{sent} < \mathrm{N}] \cdot \Bigg(\mathrm{sent} \cdot (1 - s^{\mathrm{N}-\mathrm{sent}}) \\ &\quad + (1-s) \cdot \frac{s - (\mathrm{N} - \mathrm{sent}) \cdot s^{\mathrm{N}-\mathrm{sent}} + (\mathrm{N} - \mathrm{sent} - 1) \cdot s^{\mathrm{N}-\mathrm{sent}+1}}{(1-s)^2} \Bigg) \qquad \text{(Substitute)} \\ &= [\mathrm{success} \wedge \mathrm{sent} < \mathrm{N}] \cdot \Bigg(\mathrm{sent} \cdot (1 - s^{\mathrm{N}-\mathrm{sent}}) \\ &\quad + \frac{s - (\mathrm{N} - \mathrm{sent}) \cdot s^{\mathrm{N}-\mathrm{sent}} + (\mathrm{N} - \mathrm{sent} - 1) \cdot s^{\mathrm{N}-\mathrm{sent}+1}}{1-s} \Bigg) \qquad \text{(Simplify)} \end{aligned}$$

After completing the simplification of S_1, attention shifts to S_2, which is treated analogously:

$$\begin{aligned} S_2 &= [\mathrm{success} \wedge \mathrm{sent} < \mathrm{N}] \cdot \sum_{i=0}^{\infty} [\mathrm{sent} + i + 1 = \mathrm{N}] \cdot s^{i+1} \cdot (\mathrm{sent} + i + 1) \\ &= [\mathrm{success} \wedge \mathrm{sent} < \mathrm{N}] \cdot \sum_{i=0}^{\infty} [i = \mathrm{N} - \mathrm{sent} - 1] \cdot s^{i+1} \cdot (\mathrm{sent} + i + 1) \qquad \text{(Reorder)} \\ &= [\mathrm{success} \wedge \mathrm{sent} < \mathrm{N}] \cdot s^{\mathrm{N}-\mathrm{sent}-1+1} \cdot (\mathrm{sent} + \mathrm{N} - \mathrm{sent} - 1 + 1) \qquad \text{(Restrict the sum)} \\ &= [\mathrm{success} \wedge \mathrm{sent} < \mathrm{N}] \cdot s^{\mathrm{N}-\mathrm{sent}} \cdot \mathrm{N} \qquad \text{(Simplify)} \end{aligned}$$

Finally, having simplified both S_1 and S_2, their results are combined to complete the simplification of the original supremum expression.

$$
\begin{aligned}
S_1 + S_2 &= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\text{sent} \cdot (1 - s^{\text{N}-\text{sent}}) \\
&\qquad + \frac{s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1}}{1 - s} \Bigg) \\
&\quad + [\text{success} \wedge \text{sent} < \text{N}] \cdot s^{\text{N}-\text{sent}} \cdot \text{N} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\text{sent} \cdot (1 - s^{\text{N}-\text{sent}}) \\
&\quad + \frac{s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1}}{1 - s} \\
&\quad + s^{\text{N}-\text{sent}} \cdot \text{N} \Bigg) && \text{(Distribute)} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\text{sent} - \text{sent} \cdot s^{\text{N}-\text{sent}} + \text{N} \cdot s^{\text{N}-\text{sent}} \\
&\quad + \frac{s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1}}{1 - s} \Bigg) && \text{(Distribute, Reorder)} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \Bigg(\frac{1 - s}{1 - s} \cdot (\text{sent} - \text{sent} \cdot s^{\text{N}-\text{sent}} + \text{N} \cdot s^{\text{N}-\text{sent}}) \\
&\quad + \frac{s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1}}{1 - s} \Bigg) && \text{(Multiply by 1)} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \frac{1}{1 - s} \cdot \Bigg((1 - s) \cdot (\text{sent} - \text{sent} \cdot s^{\text{N}-\text{sent}} + \text{N} \cdot s^{\text{N}-\text{sent}}) \\
&\quad + s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} \Bigg) && \text{(Distribute)} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \frac{A}{1 - s}
\end{aligned}
$$

$$
\begin{aligned}
\text{Where } A &= (1 - s) \cdot (\text{sent} - \text{sent} \cdot s^{\text{N}-\text{sent}} + \text{N} \cdot s^{\text{N}-\text{sent}}) \\
&\quad + s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} \\
&= \text{sent} - \text{sent} \cdot s^{\text{N}-\text{sent}} + \text{N} \cdot s^{\text{N}-\text{sent}} \\
&\quad - s \cdot \text{sent} + s \cdot \text{sent} \cdot s^{\text{N}-\text{sent}} - s \cdot \text{N} \cdot s^{\text{N}-\text{sent}} \\
&\quad + s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} && \text{(Distribute)} \\
&= \text{sent} - \text{sent} \cdot s^{\text{N}-\text{sent}} + \text{N} \cdot s^{\text{N}-\text{sent}} \\
&\quad - s \cdot \text{sent} + \text{sent} \cdot s^{\text{N}-\text{sent}+1} - \text{N} \cdot s^{\text{N}-\text{sent}+1} \\
&\quad + s - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} && \text{(Simplify)}
\end{aligned}
$$

... (continues on next page)

... (continued from previous page)

$$
\begin{aligned}
&= \text{sent} + (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} - (\text{N} - \text{sent}) \cdot s^{\text{N}-\text{sent}} \\
&\quad - s \cdot \text{sent} + \text{sent} \cdot s^{\text{N}-\text{sent}+1} \\
&\quad - \text{N} \cdot s^{\text{N}-\text{sent}+1} + s + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} && \text{(Distribute, Reorder)} \\
&= \text{sent} - s \cdot \text{sent} + \text{sent} \cdot s^{\text{N}-\text{sent}+1} \\
&\quad - \text{N} \cdot s^{\text{N}-\text{sent}+1} + s + (\text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} && \text{(Simplify)} \\
&= \text{sent} - s \cdot \text{sent} + s + (\text{sent} - \text{N} + \text{N} - \text{sent} - 1) \cdot s^{\text{N}-\text{sent}+1} && \text{(Distribute)} \\
&= (1 - s) \cdot \text{sent} + s - s^{\text{N}-\text{sent}+1} && \text{(Distribute, Simplify)}
\end{aligned}
$$

$$
\begin{aligned}
S_1 + S_2 &= [\text{success} \wedge \text{sent} < \text{N}] \cdot \frac{A}{1 - s} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \frac{(1 - s) \cdot \text{sent} + s - s^{\text{N}-\text{sent}+1}}{1 - s} && \text{(Substitute } A\text{)} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \left(\text{sent} + \frac{s - s^{\text{N}-\text{sent}+1}}{1 - s}\right) && \text{(Split the fraction)} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \left(\text{sent} + \frac{s(1 - s^{\text{N}-\text{sent}})}{1 - s}\right) && \text{(Distribute)}
\end{aligned}
$$

We now revisit the assumption made during the application of Equation B.2, which was introduced to avoid division by 0. However, upon examining the simplified expression, it becomes evident that when $s = 1$, the numerator of the fraction also evaluates to zero. As a result, the assumption $s < 1$ is no longer required.

We thus arrive at the simplified from of the supremum of $\text{wp}[\![\texttt{BRP}]\!](\text{sent})$:

$$
\begin{aligned}
\sup_{n \in \mathbb{N}} \Phi^n &= (S_1 + S_2) + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent} \\
&= [\text{success} \wedge \text{sent} < \text{N}] \cdot \left(\text{sent} + \frac{s(1 - s^{\text{N}-\text{sent}})}{1 - s}\right) \\
&\quad + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent}
\end{aligned}
$$

APPENDIX C

SUPREMA AS INVARIANTS

This chapter presents the proofs demonstrating that each supremum qualifies as both a wp-superinvariant and a wp-subinvariant. The methodological steps followed in the individual sections of this chapter are introduced in Section 4.3.1 and are repeated below for reference:

1. Propose a candidate invariant I (in this context, the supremum)
2. Compute $\Phi(I)$.
3. Verify that $\Phi(I) = I$ holds, to determine whether I constitutes a wp-superinvariant and a wp-subinvariant, i.e. whether $\Phi(I) = I$ holds (see Definition 10, page 12).

C.1 SendPacket

In the subsections below, the loop-characteristic function corresponding to the probabilistic program fragment is defined as follows, where g denotes the postexpectation.

$$\begin{aligned}\Phi_{\texttt{sp}}(X) &= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X) \\ &\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot g \\ \text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X) &= p \cdot X[\text{success} := \mathit{true}] + (1-p) \cdot X[\text{failed} := \text{failed} + 1]\end{aligned}$$

C.1.1 Probability of Success

The loop-characteristic function of the while-loop of $\text{wp}[\![\texttt{sendPacket}]\!]([\text{success}])$ is the following:

$$\Phi_{\texttt{sp}}(X) = [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X) + [\text{success}]$$

Step 1: Propose candidate invariant I

$$I = \sup_{n\in\mathbb{N}} \Phi^n = [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot (1 - (1-p)^{\text{MAX}-\text{failed}}) + [\text{success}]$$

Step 2: Compute $\Phi_{\texttt{sp}}(I)$

$$\begin{aligned}\text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](I) = p &\\ + [\text{failed} + 1 < \text{MAX} \wedge \neg\text{success}] &\cdot ((1-p) - (1-p)^{\text{MAX}-\text{failed}}) \\ + (1-p) &\cdot [\text{success}]\end{aligned}$$

$$\begin{aligned}\Phi_{\texttt{sp}}(I) = {} & [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot p \\ & + [\text{failed} + 1 < \text{MAX} \wedge \neg\text{success}] \cdot (1 - p - (1-p)^{\text{MAX}-\text{failed}}) \\ & + [\text{success}]\end{aligned}$$

Step 3: Compare $\Phi_{\texttt{sp}}(I)$ and I

By performing a case analysis on the inequality between the variables `MAX` and `failed`, it is shown that I satisfies both the wp-superinvariant and wp-subinvariant conditions, as $\Phi_{\texttt{sp}}(I) = I$ in all cases.

- **Case 1**: failed $\geq$ MAX.

$$\begin{aligned} I &= [\mathit{false} \wedge \neg\text{success}] \cdot (1 - (1-p)^{\text{MAX}-\text{failed}}) + [\text{success}] && \text{(Apply case)} \\ &= [\text{success}] && \text{(Simplify)} \end{aligned}$$

$$\begin{aligned} \Phi_{\texttt{sp}}(I) = {} & [\mathit{false} \wedge \neg\text{success}] \cdot p \\ & + [\mathit{false} \wedge \neg\text{success}] \cdot (1 - p - (1-p)^{\text{MAX}-\text{failed}}) \\ & + [\text{success}] && \text{(Apply case)} \\ = {} & [\text{success}] && \text{(Simplify)} \\ = {} & I \end{aligned}$$

- **Case 2**: failed $<$ MAX $\wedge$ failed $+ 1 \geq$ MAX $\implies$ failed $+ 1 =$ MAX

$$\begin{aligned} I = {} & [\text{failed} < \text{failed} + 1 \wedge \neg\text{success}] \cdot (1 - (1-p)^{\text{failed}+1-\text{failed}}) \\ & + [\text{success}] && \text{(Apply case)} \\ = {} & [\mathit{true} \wedge \neg\text{success}] \cdot (1 - 1 + p) + [\text{success}] && \text{(Simplify)} \\ = {} & [\neg\text{success}] \cdot p + [\text{success}] && \text{(Simplify)} \end{aligned}$$

$$\begin{aligned} \Phi_{\texttt{sp}}(I) = {} & [\text{failed} < \text{failed} + 1 \wedge \neg\text{success}] \cdot p \\ & + [\text{failed} + 1 < \text{failed} + 1 \wedge \neg\text{success}] \cdot (1 - p - (1-p)^{\text{failed}+1-\text{failed}}) \\ & + [\text{success}] && \text{(Apply case)} \\ = {} & [\mathit{true} \wedge \neg\text{success}] \cdot p \\ & + [\mathit{false} \wedge \neg\text{success}] \cdot (1 - p - (1-p) \\ & + [\text{success}] && \text{(Simplify)} \\ = {} & [\neg\text{success}] \cdot p + [\text{success}] && \text{(Simplify)} \\ = {} & I \end{aligned}$$

- **Case 3**: failed $<$ MAX $\wedge$ failed $+ 1 <$ MAX.

$$\begin{aligned} I &= [\mathit{true} \wedge \neg\text{success}] \cdot (1 - (1-p)^{\text{MAX}-\text{failed}}) + [\text{success}] && \text{(Apply case)} \\ &= [\neg\text{success}] \cdot (1 - (1-p)^{\text{MAX}-\text{failed}}) + [\text{success}] && \text{(Simplify)} \end{aligned}$$

$$
\begin{aligned}
\Phi_{\mathtt{sp}}(I) &= [\mathit{true} \wedge \neg\text{success}] \cdot p \\
&\quad + [\mathit{true} \wedge \neg\text{success}] \cdot (1 - p - (1-p)^{\text{MAX}-\text{failed}}) \\
&\quad + [\text{success}] && \text{(Apply case)} \\
&= [\neg\text{success}] \cdot p + [\neg\text{success}] \cdot (1 - p - (1-p)^{\text{MAX}-\text{failed}}) \\
&\quad + [\text{success}] && \text{(Simplify)} \\
&= [\neg\text{success}] \cdot (p + 1 - p - (1-p)^{\text{MAX}-\text{failed}}) + [\text{success}] && \text{(Distribute)} \\
&= [\neg\text{success}] \cdot (1 - (1-p)^{\text{MAX}-\text{failed}}) + [\text{success}] && \text{(Simplify)} \\
&= I
\end{aligned}
$$

Therefore, the supremum qualifies as both a wp-superinvariant and a wp-subinvariant.

C.1.2 Expected Number of Failures

The loop-characteristic function of the while-loop of wp⟦`sendPacket`⟧(failed) is the following:

$$
\begin{aligned}
\Phi_{\mathtt{sp}}(X) &= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \text{wp}[\![\mathtt{body}_{\mathtt{sp}}]\!](X) \\
&\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed}
\end{aligned}
$$

Step 1: Propose candidate invariant I

$$
\begin{aligned}
I &= \sup_{n \in \mathbb{N}} \Phi^n \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \left(\text{failed} + \frac{(1-p) - (1-p)^{\text{MAX}-\text{failed}+1}}{p} \right) \\
&\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed}
\end{aligned}
$$

Step 2: Compute $\Phi_{\mathtt{sp}}(I)$

$$
\begin{aligned}
\text{wp}[\![\mathtt{body}_{\mathtt{sp}}]\!](I) &= p \cdot \text{failed} \\
&\quad + [\text{failed} + 1 < \text{MAX} \wedge \neg\text{success}] \cdot (1-p) \cdot \left(\text{failed} + 1 \right. \\
&\quad \left. + \frac{1 - p - (1-p)^{\text{MAX}-\text{failed}}}{p} \right) \\
&\quad + [\text{failed} + 1 \geq \text{MAX} \vee \text{success}] \cdot (1-p) \cdot (\text{failed} + 1)
\end{aligned}
$$

$$
\begin{aligned}
\Phi_{\mathtt{sp}}(I) &= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot p \cdot \text{failed} \\
&\quad + [\text{failed} + 1 < \text{MAX} \wedge \neg\text{success}] \cdot (1-p) \cdot \left(\text{failed} + 1 \right. \\
&\quad \left. + \frac{1 - p - (1-p)^{\text{MAX}-\text{failed}}}{p} \right) \\
&\quad + [\text{failed} + 1 = \text{MAX} \wedge \neg\text{success}] \cdot (1-p) \cdot (\text{failed} + 1) \\
&\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed} \\
&\ldots \text{(continues on next page)}
\end{aligned}
$$

... (continued from previous page)

$$
\begin{aligned}
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot p \cdot \text{failed} \\
&\quad + [\text{failed} + 1 \leq \text{MAX} \wedge \neg\text{success}] \cdot (1-p) \cdot \Big(\text{failed} + 1 \\
&\qquad + \frac{1 - p - (1-p)^{\text{MAX}-\text{failed}}}{p}\Big) \\
&\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed} && \text{(Simplify)} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot p \cdot \text{failed} \\
&\quad + [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot (1-p) \cdot \Big(\text{failed} + 1 \\
&\qquad + \frac{1 - p - (1-p)^{\text{MAX}-\text{failed}}}{p}\Big) \\
&\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed} && \text{(Simplify)} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \Big(p \cdot \text{failed} + (1-p) \cdot \Big(\text{failed} + 1 \\
&\qquad + \frac{1 - p - (1-p)^{\text{MAX}-\text{failed}}}{p}\Big)\Big) \\
&\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed} && \text{(Distribute)} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \Big(p \cdot \text{failed} + (1-p) \cdot \text{failed} + (1-p) \\
&\qquad + (1-p) \cdot \frac{1 - p - (1-p)^{\text{MAX}-\text{failed}}}{p}\Big) \\
&\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed} && \text{(Distribute)} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \Big(p \cdot \text{failed} + (1-p) \cdot \text{failed} + 1 - p \\
&\qquad + (1-p) \cdot \frac{-p}{p} + (1-p) \cdot \frac{1 - (1-p)^{\text{MAX}-\text{failed}}}{p}\Big) \\
&\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed} && \text{(Factor out } p\text{)} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \Big(p \cdot \text{failed} + (1-p) \cdot \text{failed} + 1 - p \\
&\qquad - (1-p) + (1-p) \cdot \frac{1 - (1-p)^{\text{MAX}-\text{failed}}}{p}\Big) \\
&\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed} && \text{(Simplify)} \\
&= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \Big(p \cdot \text{failed} + \text{failed} - p \cdot \text{failed} + 1 - p \\
&\qquad - 1 + p + \frac{(1-p) - (1-p) \cdot (1-p)^{\text{MAX}-\text{failed}}}{p}\Big) \\
&\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed} && \text{(Distribute)}
\end{aligned}
$$

... (continues on next page)

... (continued from previous page)

$$= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \left(\text{failed} + \frac{1-p-(1-p)^{\text{MAX}-\text{failed}+1}}{p}\right)$$
$$+ [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed} \qquad \text{(Simplify)}$$

Step 3: Compare $\Phi_{\texttt{sp}}(I)$ and I

The fully simplified expression for $\Phi_{\texttt{sp}}(I)$ is identical to the proposed invariant I. Therefore, $\Phi_{\texttt{sp}}(I) = I$ holds, and the supremum qualifies as both a wp-superinvariant and a wp-subinvariant.

C.2 BRP

In the subsections below, the loop-characteristic function corresponding to the probabilistic program fragment is defined as follows, where g denotes the postexpectation.

$$\Phi_{\texttt{brp}}(X) = [\text{sent} < \text{N} \wedge \text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot g$$
$$\text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) = s \cdot X[\text{sent} := \text{sent} + 1; \text{totalFailed} := \text{totalFailed} + f; \text{success} := \mathit{true}]$$
$$+ (1-s) \cdot X[\text{totalFailed} := \text{totalFailed} + f; \text{success} := \mathit{false}]$$

C.2.1 Probability of Success

The loop-characteristic function of the while-loop of $\text{wp}[\![\texttt{BRP}]\!]([\text{success}])$ is the following:

$$\Phi_{\texttt{brp}}(X) = [\text{sent} < \text{N} \wedge \text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) + [\text{sent} \geq \text{N} \wedge \text{success}]$$

Step 1: Propose candidate invariant I

$$I = \sup_{n \in \mathbb{N}} \Phi^n = [\text{sent} < \text{N} \wedge \text{success}] \cdot s^{\text{N}-\text{sent}} + [\text{sent} \geq \text{N} \wedge \text{success}]$$

Step 2: Compute $\Phi_{brp}(I)$

$$\begin{aligned}\text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](I) &= [\text{sent}+1 < \text{N} \wedge \text{success}] \cdot s \cdot s^{\text{N}-\text{sent}-1} \\ &\quad + [\text{sent}+1 \geq \text{N} \wedge \text{success}] \cdot s \\ &= [\text{sent}+1 < \text{N} \wedge \text{success}] \cdot s^{\text{N}-\text{sent}} \\ &\quad + [\text{sent}+1 \geq \text{N} \wedge \text{success}] \cdot s\end{aligned}$$

$$\begin{aligned}\Phi_{brp}(I) &= [\text{sent}+1 < \text{N} \wedge \text{success}] \cdot s^{\text{N}-\text{sent}} \\ &\quad + [\text{sent}+1 = \text{N} \wedge \text{success}] \cdot s \\ &\quad + [\text{sent} \geq \text{N} \wedge \text{success}] \\ &= [\text{sent}+1 \leq \text{N} \wedge \text{success}] \cdot s^{\text{N}-\text{sent}} + [\text{sent} \geq \text{N} \wedge \text{success}] \qquad \text{(Simplify)} \\ &= [\text{sent} < \text{N} \wedge \text{success}] \cdot s^{\text{N}-\text{sent}} + [\text{sent} \geq \text{N} \wedge \text{success}] \qquad \text{(Simplify)}\end{aligned}$$

Step 3: Compare $\Phi_{\texttt{brp}}(I)$ and I

The fully simplified expression for $\Phi_{\texttt{brp}}(I)$ is identical to the proposed invariant I. Therefore, $\Phi_{\texttt{brp}}(I) = I$ holds, and the supremum qualifies as both a wp-superinvariant and a wp-subinvariant.

C.2.2 Expected Number of Failures

The loop-characteristic function of the while-loop of wp$[\![\texttt{BRP}]\!]$(totalFailed) is the following:

$$\begin{aligned}\Phi_{\texttt{brp}}(X) = {} & [\text{sent} < \text{N} \wedge \text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) \\ & + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed}\end{aligned}$$

Step 1: Propose candidate invariant I

$$\begin{aligned}I = \sup_{n\in\mathbb{N}}\Phi^n = {} & [\text{success} \wedge \text{sent} < \text{N}] \cdot \left(\text{totalFailed} + f \cdot \frac{1 - s^{\text{N}-\text{sent}}}{1-s}\right) \\ & + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{totalFailed}\end{aligned}$$

Step 2: Compute $\Phi_{brp}(I)$

$$\begin{aligned}\text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](I) = {} & [\text{sent} + 1 < \text{N}] \cdot s \cdot \left(\text{totalFailed} + f + f \cdot \frac{1 - s^{\text{N}-\text{sent}-1}}{1-s}\right) \\ & + [\text{sent} + 1 \geq \text{N}] \cdot s \cdot (\text{totalFailed} + f) \\ & + (1-s) \cdot (\text{totalFailed} + f)\end{aligned}$$

$$\begin{aligned}\Phi_{brp}(I) = {} & [\text{sent} + 1 < \text{N} \wedge \text{success}] \cdot s \cdot \left(\text{totalFailed} + f + f \cdot \frac{1 - s^{\text{N}-\text{sent}-1}}{1-s}\right) \\ & + [\text{sent} + 1 = \text{N} \wedge \text{success}] \cdot s \cdot (\text{totalFailed} + f) \\ & + [\text{sent} < \text{N} \wedge \text{success}] \cdot (1-s) \cdot (\text{totalFailed} + f) \\ & + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \\ = {} & [\text{sent} + 1 \leq \text{N} \wedge \text{success}] \cdot s \cdot \left(\text{totalFailed} + f + f \cdot \frac{1 - s^{\text{N}-\text{sent}-1}}{1-s}\right) \\ & + [\text{sent} < \text{N} \wedge \text{success}] \cdot (1-s) \cdot (\text{totalFailed} + f) \\ & + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \qquad \text{(Simplify)} \\ = {} & [\text{sent} < \text{N} \wedge \text{success}] \cdot s \cdot \left(\text{totalFailed} + f + f \cdot \frac{1 - s^{\text{N}-\text{sent}-1}}{1-s}\right) \\ & + [\text{sent} < \text{N} \wedge \text{success}] \cdot (1-s) \cdot (\text{totalFailed} + f) \\ & + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \qquad \text{(Simplify)}\end{aligned}$$

. . . (continues on next page)

... (continued from previous page)

$$= [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(s \cdot \left(\text{totalFailed} + f + f \cdot \frac{1 - s^{\text{N}-\text{sent}-1}}{1 - s} \right) + (1 - s) \cdot (\text{totalFailed} + f) \right) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \quad \text{(Simplify)}$$

$$= [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(s \cdot \text{totalFailed} + s \cdot f + s \cdot f \cdot \frac{1 - s^{\text{N}-\text{sent}-1}}{1 - s} + (1 - s) \cdot \text{totalFailed} + (1 - s) \cdot f \right) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \quad \text{(Distribute)}$$

$$= [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(s \cdot \text{totalFailed} + s \cdot f + s \cdot f \cdot \frac{1 - s^{\text{N}-\text{sent}-1}}{1 - s} + \text{totalFailed} - s \cdot \text{totalFailed} + f - s \cdot f \right) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \quad \text{(Distribute)}$$

$$= [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(s \cdot \text{totalFailed} - s \cdot \text{totalFailed} + s \cdot f - s \cdot f + \text{totalFailed} + f + s \cdot f \cdot \frac{1 - s^{\text{N}-\text{sent}-1}}{1 - s} \right) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \quad \text{(Reorder)}$$

$$= [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(\text{totalFailed} + f + s \cdot f \cdot \frac{1 - s^{\text{N}-\text{sent}-1}}{1 - s} \right) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \quad \text{(Simplify)}$$

$$= [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(\text{totalFailed} + f + f \cdot \frac{s - s^{\text{N}-\text{sent}}}{1 - s} \right) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \quad \text{(Simplify)}$$

$$= [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(\text{totalFailed} + f \cdot \frac{1 - s}{1 - s} + f \cdot \frac{s - s^{\text{N}-\text{sent}}}{1 - s} \right) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \quad \text{(Multiply by 1)}$$

$$= [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(\text{totalFailed} + f \cdot \frac{1 - s + s - s^{\text{N}-\text{sent}}}{1 - s} \right) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \quad \text{(Distribute)}$$

$$= [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(\text{totalFailed} + f \cdot \frac{1 - s^{\text{N}-\text{sent}}}{1 - s} \right) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \quad \text{(Simplify)}$$

Step 3: Compare $\Phi_{\texttt{brp}}(I)$ and I

The fully simplified expression for $\Phi_{\texttt{brp}}(I)$ is identical to the proposed invariant I. Therefore, $\Phi_{\texttt{brp}}(I) = I$ holds, and the supremum qualifies as both a wp-superinvariant and a wp-subinvariant.

C.2.3 Expected Number of Sent Packets

The loop-characteristic function of the while-loop of $\text{wp}[\![\texttt{BRP}]\!](\text{sent})$ is the following:

$$\Phi_{\texttt{brp}}(X) = [\text{sent} < \text{N} \wedge \text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{sent}$$

Step 1: Propose candidate invariant I

$$\begin{aligned} I = \sup_{n\in\mathbb{N}} \Phi^n = {} & [\text{success} \wedge \text{sent} < \text{N}] \cdot \left(\text{sent} + \frac{s(1 - s^{\text{N}-\text{sent}})}{1-s}\right) \\ & + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent} \end{aligned}$$

Step 2: Compute $\Phi_{brp}(I)$

$$\begin{aligned} \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](I) = {} & [\text{sent} + 1 < \text{N}] \cdot s \cdot \left(\text{sent} + 1 + \frac{s(1 - s^{\text{N}-\text{sent}-1})}{1-s}\right) \\ & + [\text{sent} + 1 \geq \text{N}] \cdot s \cdot (\text{sent} + 1) \\ & + (1-s) \cdot \text{sent} \end{aligned}$$

$$\begin{aligned} \Phi_{brp}(I) = {} & [\text{sent} + 1 < \text{N} \wedge \text{success}] \cdot s \cdot \left(\text{sent} + 1 + \frac{s(1 - s^{\text{N}-\text{sent}-1})}{1-s}\right) \\ & + [\text{sent} + 1 = \text{N} \wedge \text{success}] \cdot s \cdot (\text{sent} + 1) \\ & + [\text{sent} < \text{N} \wedge \text{success}] \cdot (1-s) \cdot \text{sent} \\ & + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent} \\ = {} & [\text{sent} + 1 \leq \text{N} \wedge \text{success}] \cdot s \cdot \left(\text{sent} + 1 + \frac{s(1 - s^{\text{N}-\text{sent}-1})}{1-s}\right) \\ & + [\text{sent} < \text{N} \wedge \text{success}] \cdot (1-s) \cdot \text{sent} \\ & + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent} && \text{(Simplify)} \\ = {} & [\text{sent} < \text{N} \wedge \text{success}] \cdot s \cdot \left(\text{sent} + 1 + \frac{s(1 - s^{\text{N}-\text{sent}-1})}{1-s}\right) \\ & + [\text{sent} < \text{N} \wedge \text{success}] \cdot (1-s) \cdot \text{sent} \\ & + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent} && \text{(Simplify)} \\ = {} & [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(s \cdot \left(\text{sent} + 1 + \frac{s(1 - s^{\text{N}-\text{sent}-1})}{1-s}\right)\right. \\ & \left. + (1-s) \cdot \text{sent}\right) + [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent} && \text{(Distribute)} \end{aligned}$$

... (continues on next page)

... (continued from previous page)

$$= [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(s \cdot \left(\text{sent} + 1 + \frac{s - s^{\text{N}-\text{sent}}}{1-s} \right) + \text{sent} - s \cdot \text{sent} \right)$$
$$+ [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent} \qquad \text{(Distribute)}$$

$$= [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(s \cdot \text{sent} + s + s \cdot \frac{s - s^{\text{N}-\text{sent}}}{1-s} + \text{sent} - s \cdot \text{sent} \right)$$
$$+ [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent} \qquad \text{(Distribute)}$$

$$= [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(\text{sent} + s + s \cdot \frac{s - s^{\text{N}-\text{sent}}}{1-s} - s \cdot \text{sent} + s \cdot \text{sent} \right)$$
$$+ [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent} \qquad \text{(Reorder)}$$

$$= [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(\text{sent} + s \cdot \frac{1-s}{1-s} + s \cdot \frac{s - s^{\text{N}-\text{sent}}}{1-s} \right)$$
$$+ [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent} \qquad \text{(Multiply by 1, Simplify)}$$

$$= [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(\text{sent} + s \cdot \frac{1 - s + s - s^{\text{N}-\text{sent}}}{1-s} \right)$$
$$+ [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent} \qquad \text{(Distribute)}$$

$$= [\text{sent} < \text{N} \wedge \text{success}] \cdot \left(\text{sent} + s \cdot \frac{1 - s^{\text{N}-\text{sent}}}{1-s} \right)$$
$$+ [\neg\text{success} \vee \text{sent} \geq \text{N}] \cdot \text{sent} \qquad \text{(Simplify)}$$

Step 3: Compare $\Phi_{\texttt{brp}}(I)$ and I

The fully simplified expression for $\Phi_{\texttt{brp}}(I)$ is identical to the proposed invariant I. Therefore, $\Phi_{\texttt{brp}}(I) = I$ holds, and the supremum qualifies as both a wp-superinvariant and a wp-subinvariant.

APPENDIX D

SUPERINVARIANTS

This chapter presents the proofs demonstrating that the proposed invariants constitute wp-superinvariants. The methodology applied in the subsequent sections follows the steps outlined in Section 4.3.1, and is reiterated below for clarity:

1. Propose a candidate superinvariant I
2. Compute $\Phi(I)$.
3. Compare $\Phi(I)$ with I to determine whether it qualifies as a wp-superinvariant, i.e. whether $\Phi(I) \sqsubseteq I$ holds (see Definition 10, page 12).

D.1 SendPacket

The equation for the loop-characteristic function in this section is provided below, where g denotes the postexpectation:

$$\begin{aligned}\Phi_{\texttt{sp}}(X) &= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X) \\ &\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot g \\ \text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X) &= p \cdot X[\text{success} := \mathit{true}] + (1-p) \cdot X[\text{failed} := \text{failed} + 1]\end{aligned}$$

D.1.1 Expected Number of Failures

The loop-characteristic function of the while-loop of $\text{wp}[\![\texttt{sendPacket}]\!](\text{failed})$ is as follows:

$$\begin{aligned}\Phi_{\texttt{sp}}(X) &= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{sp}}]\!](X) \\ &\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed}\end{aligned}$$

Step 1: Propose candidate superinvariant I

$$\begin{aligned}I &= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot (\text{failed} + (1-p) \cdot (\text{MAX} - \text{failed})) \\ &\quad + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed}\end{aligned}$$

Step 2: Compute $\Phi_{\texttt{sp}}(I)$

$$\begin{aligned}
\mathrm{wp}[\![\texttt{body}_{\texttt{sp}}]\!](I) &= p \cdot \mathrm{failed} \\
&\quad + [\mathrm{failed} + 1 < \mathrm{MAX} \wedge \neg\mathrm{success}] \cdot (1-p) \cdot \Big(\mathrm{failed} + 1 \\
&\qquad + (1-p) \cdot (\mathrm{MAX} - \mathrm{failed} - 1)\Big) \\
&\quad + [\mathrm{failed} + 1 \geq \mathrm{MAX} \vee \mathrm{success}] \cdot (1-p) \cdot (\mathrm{failed} + 1)
\end{aligned}$$

$$\begin{aligned}
\Phi_{\texttt{sp}}(I) &= [\mathrm{failed} < \mathrm{MAX} \wedge \neg\mathrm{success}] \cdot p \cdot \mathrm{failed} \\
&\quad + [\mathrm{failed} + 1 < \mathrm{MAX} \wedge \neg\mathrm{success}] \cdot (1-p) \cdot \Big(\mathrm{failed} + 1 \\
&\qquad + (1-p) \cdot (\mathrm{MAX} - \mathrm{failed} - 1)\Big) \\
&\quad + [\mathrm{failed} + 1 = \mathrm{MAX} \wedge \neg\mathrm{success}] \cdot (1-p) \cdot (\mathrm{failed} + 1) \\
&\quad + [\mathrm{failed} \geq \mathrm{MAX} \vee \mathrm{success}] \cdot \mathrm{failed} \\
&= [\mathrm{failed} < \mathrm{MAX} \wedge \neg\mathrm{success}] \cdot p \cdot \mathrm{failed} \\
&\quad + [\mathrm{failed} + 1 \leq \mathrm{MAX} \wedge \neg\mathrm{success}] \cdot (1-p) \cdot \Big(\mathrm{failed} + 1 \\
&\qquad + (1-p) \cdot (\mathrm{MAX} - \mathrm{failed} - 1)\Big) \\
&\quad + [\mathrm{failed} \geq \mathrm{MAX} \vee \mathrm{success}] \cdot \mathrm{failed} && \text{(Simplify)} \\
&= [\mathrm{failed} < \mathrm{MAX} \wedge \neg\mathrm{success}] \cdot p \cdot \mathrm{failed} \\
&\quad + [\mathrm{failed} < \mathrm{MAX} \wedge \neg\mathrm{success}] \cdot (1-p) \cdot \Big(\mathrm{failed} + 1 \\
&\qquad + (1-p) \cdot (\mathrm{MAX} - \mathrm{failed} - 1)\Big) \\
&\quad + [\mathrm{failed} \geq \mathrm{MAX} \vee \mathrm{success}] \cdot \mathrm{failed} && \text{(Simplify)} \\
&= [\mathrm{failed} < \mathrm{MAX} \wedge \neg\mathrm{success}] \cdot \Big(p \cdot \mathrm{failed} + (1-p) \cdot \Big(\mathrm{failed} + 1 \\
&\qquad + (1-p) \cdot (\mathrm{MAX} - \mathrm{failed} - 1)\Big)\Big) \\
&\quad + [\mathrm{failed} \geq \mathrm{MAX} \vee \mathrm{success}] \cdot \mathrm{failed} && \text{(Distribute)} \\
&= [\mathrm{failed} < \mathrm{MAX} \wedge \neg\mathrm{success}] \cdot \Big(p \cdot \mathrm{failed} + (1-p) \cdot \mathrm{failed} + (1-p) \\
&\qquad + (1-p) \cdot (1-p) \cdot (\mathrm{MAX} - \mathrm{failed} - 1)\Big) \\
&\quad + [\mathrm{failed} \geq \mathrm{MAX} \vee \mathrm{success}] \cdot \mathrm{failed} && \text{(Distribute)} \\
&= [\mathrm{failed} < \mathrm{MAX} \wedge \neg\mathrm{success}] \cdot \Big(p \cdot \mathrm{failed} + \mathrm{failed} - p \cdot \mathrm{failed} + (1-p) \\
&\qquad + (1-p) \cdot (1-p) \cdot (\mathrm{MAX} - \mathrm{failed} - 1)\Big) \\
&\quad + [\mathrm{failed} \geq \mathrm{MAX} \vee \mathrm{success}] \cdot \mathrm{failed} && \text{(Distribute)} \\
&= [\mathrm{failed} < \mathrm{MAX} \wedge \neg\mathrm{success}] \cdot \Big(\mathrm{failed} + (1-p) \\
&\qquad + (1-p) \cdot (1-p) \cdot (\mathrm{MAX} - \mathrm{failed} - 1)\Big) \\
&\quad + [\mathrm{failed} \geq \mathrm{MAX} \vee \mathrm{success}] \cdot \mathrm{failed} && \text{(Simplify)}
\end{aligned}$$

$$= [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \Big(\text{failed} + (1-p) \cdot \big(1 + (1-p) \cdot (\text{MAX} - \text{failed} - 1)\big)\Big) + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed} \qquad \text{(Distribute)}$$

Step 3: Compare $\Phi_{\texttt{sp}}(I)$ and I

The simplified expression of $\Phi_{\texttt{sp}}(I)$ closely resembles the candidate invariant I:

$$I = [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot (\text{failed} + (1-p) \cdot (\text{MAX} - \text{failed})) + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed}$$

$$\Phi_{\texttt{sp}}(I) = [\text{failed} < \text{MAX} \wedge \neg\text{success}] \cdot \Big(\text{failed} + (1-p) \cdot \big(1 + (1-p) \cdot (\text{MAX} - \text{failed} - 1)\big)\Big) + [\text{failed} \geq \text{MAX} \vee \text{success}] \cdot \text{failed}$$

To assess the relationship between $\Phi_{\texttt{sp}}(I)$ and I, we compare the corresponding highlighted components:

$$\begin{aligned} & \text{MAX} - \text{failed} > \text{MAX} - \text{failed} - 1 \\ \implies\ & \text{MAX} - \text{failed} > (1-p) \cdot (\text{MAX} - \text{failed} - 1) && (0 \leq (1-p) \leq 1) \\ \implies\ & \text{MAX} - \text{failed} \geq (1-p) \cdot (\text{MAX} - \text{failed} - 1) + 1 \\ \implies\ & (1-p) \cdot (\text{MAX} - \text{failed}) \geq (1-p) \cdot ((1-p) \cdot (\text{MAX} - \text{failed} - 1) + 1) && ((1-p) \geq 0) \\ \implies\ & I \geq \Phi_{\texttt{sp}}(I) \end{aligned}$$

Based on this comparison, we conclude that $\Phi_{\texttt{sp}}(I) \sqsubseteq I$ holds, and therefore I is indeed a wp-superinvariant.

D.2 BRP

The equation for the loop-characteristic function in this section is provided below, where g denotes the postexpectation:

$$\begin{aligned} \Phi_{\texttt{brp}}(X) &= [\text{sent} < \text{N} \wedge \text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot g \\ \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) &= s \cdot X[\text{sent} := \text{sent} + 1; \text{totalFailed} := \text{totalFailed} + f; \text{success} := \mathit{true}] \\ &\quad + (1-s) \cdot X[\text{totalFailed} := \text{totalFailed} + f; \text{success} := \mathit{false}] \end{aligned}$$

D.2.1 Probability of Success

The loop-characteristic function of the while-loop of $\text{wp}[\![\texttt{BRP}]\!]([\text{success}])$ is as follows:

$$\Phi_{\texttt{brp}}(X) = [\text{sent} < \text{N} \wedge \text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) + [\text{sent} \geq \text{N} \wedge \text{success}]$$

Step 1: Propose candidate superinvariant I

$$I = [\text{success}] \cdot s^{\text{N}-\text{sent}} + [\text{success}]$$

Step 2: Compute $\Phi_{\texttt{brp}}(I)$

$$\begin{aligned}\text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](I) &= s \cdot s^{\text{N}-\text{sent}-1} + s \\ &= s^{\text{N}-\text{sent}} + s\end{aligned}$$

$$\begin{aligned}\Phi_{\texttt{brp}}(I) = &[\text{sent} < \text{N} \wedge \text{success}] \cdot (s^{\text{N}-\text{sent}} + s) \\ &+ [\text{sent} \geq \text{N} \wedge \text{success}]\end{aligned}$$

Step 3: Compare $\Phi_{\texttt{brp}}(I)$ and I

By performing a case analysis on the inequality between the variables `N` and `sent`, it is shown that I satisfies the wp-superinvariant conditions, as $\Phi_{\texttt{brp}}(I) \leq I$ in all cases.

- **Case 1**: sent $\geq$ N.

$$\begin{aligned}\Phi_{\texttt{brp}}(I) &= [\mathit{false} \wedge \text{success}] \cdot (s^{\text{N}-\text{sent}} + s) \\ &\quad + [\mathit{true} \wedge \text{success}] && \text{(Apply case)} \\ &= [\text{success}] && \text{(Simplify)} \\ &\sqsubseteq I\end{aligned}$$

- **Case 2**: sent $<$ N.

$$\begin{aligned}\Phi_{\texttt{brp}}(I) &= [\mathit{true} \wedge \text{success}] \cdot (s^{\text{N}-\text{sent}} + s) \\ &\quad + [\mathit{false} \wedge \text{success}] && \text{(Apply case)} \\ &= [\text{success}] \cdot (s^{\text{N}-\text{sent}} + s) && \text{(Simplify)} \\ &= [\text{success}] \cdot s^{\text{N}-\text{sent}} + [\text{success}] \cdot s && \text{(Distribute)} \\ &\sqsubseteq I && (s \leq 1)\end{aligned}$$

Therefore, the candidate invariant qualifies as a wp-superinvariant.

D.2.2 Expected Number of Failures

The loop-characteristic function of the while-loop of wp$[\![\texttt{BRP}]\!]$(totalFailed) is as follows:

$$\begin{aligned}\Phi_{\texttt{brp}}(X) = &[\text{sent} < \text{N} \wedge \text{success}] \cdot \text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) \\ &+ [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed}\end{aligned}$$

Step 1: Propose candidate superinvariant I

$$\begin{aligned}I = &[\text{sent} < \text{N} \wedge \text{success}] \cdot (\text{totalFailed} + f \cdot (\text{N} - \text{sent})) \\ &+ [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed}\end{aligned}$$

Step 2: Compute $\Phi_{\texttt{brp}}(I)$

$$\begin{aligned}
\text{wp}[\![\texttt{body}_{\texttt{brp}}]\!](I) &= [\text{sent} + 1 < \text{N}] \cdot s \cdot (\text{totalFailed} + f + f \cdot (\text{N} - \text{sent} - 1)) \\
&\quad + [\text{sent} + 1 \geq \text{N}] \cdot s \cdot (\text{totalFailed} + f) \\
&\quad + (1 - s) \cdot (\text{totalFailed} + f)
\end{aligned}$$

$$\begin{aligned}
\Phi_{\texttt{brp}}(I) &= [\text{sent} + 1 < \text{N} \wedge \text{success}] \cdot s \cdot (\text{totalFailed} + f + f \cdot (\text{N} - \text{sent} - 1)) \\
&\quad + [\text{sent} + 1 = \text{N} \wedge \text{success}] \cdot s \cdot (\text{totalFailed} + f) \\
&\quad + [\text{sent} < \text{N} \wedge \text{success}] \cdot (1 - s) \cdot (\text{totalFailed} + f) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} \\
&= [\text{sent} + 1 \leq \text{N} \wedge \text{success}] \cdot s \cdot (\text{totalFailed} + f + f \cdot (\text{N} - \text{sent} - 1)) \\
&\quad + [\text{sent} < \text{N} \wedge \text{success}] \cdot (1 - s) \cdot (\text{totalFailed} + f) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} && \text{(Simplify)} \\
&= [\text{sent} < \text{N} \wedge \text{success}] \cdot s \cdot (\text{totalFailed} + f + f \cdot (\text{N} - \text{sent} - 1)) \\
&\quad + [\text{sent} < \text{N} \wedge \text{success}] \cdot (1 - s) \cdot (\text{totalFailed} + f) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} && \text{(Simplify)} \\
&= [\text{sent} < \text{N} \wedge \text{success}] \cdot \Big(s \cdot (\text{totalFailed} + f + f \cdot (\text{N} - \text{sent} - 1)) \\
&\qquad + (1 - s) \cdot (\text{totalFailed} + f)\Big) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} && \text{(Distribute)} \\
&= [\text{sent} < \text{N} \wedge \text{success}] \cdot \Big(s \cdot \text{totalFailed} + s \cdot f + s \cdot f \cdot (\text{N} - \text{sent} - 1) \\
&\qquad + \text{totalFailed} + f - s \cdot \text{totalFailed} - s \cdot f\Big) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} && \text{(Distribute)} \\
&= [\text{sent} < \text{N} \wedge \text{success}] \cdot \Big(s \cdot \text{totalFailed} - s \cdot \text{totalFailed} + s \cdot f - s \cdot f \\
&\qquad + \text{totalFailed} + s \cdot f \cdot (\text{N} - \text{sent} - 1) + f\Big) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} && \text{(Reorder)} \\
&= [\text{sent} < \text{N} \wedge \text{success}] \cdot \Big(\text{totalFailed} + s \cdot f \cdot (\text{N} - \text{sent} - 1) + f\Big) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} && \text{(Simplify)} \\
&= [\text{sent} < \text{N} \wedge \text{success}] \cdot \Big(\text{totalFailed} + f \cdot \Big(s \cdot (\text{N} - \text{sent} - 1) + 1\Big)\Big) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed} && \text{(Distribute)}
\end{aligned}$$

Step 3: Compare $\Phi_{\texttt{brp}}(I)$ and I

The simplified expression of $\Phi_{\texttt{brp}}(I)$ closely resembles the candidate invariant I:

$$\begin{aligned}
I &= [\text{sent} < \text{N} \wedge \text{success}] \cdot (\text{totalFailed} + f \cdot (\text{N} - \text{sent})) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed}
\end{aligned}$$

$$\begin{aligned}
\Phi_{\texttt{brp}}(I) &= [\text{sent} < \text{N} \wedge \text{success}] \cdot \Big(\text{totalFailed} + f \cdot \Big(s \cdot (\text{N} - \text{sent} - 1) + 1\Big)\Big) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{totalFailed}
\end{aligned}$$

To assess the relationship between $\Phi_{\texttt{brp}}(I)$ and I, we compare the corresponding highlighted components:

$$\begin{aligned}
& \mathrm{N} - \mathrm{sent} > \mathrm{N} - \mathrm{sent} - 1 \\
\implies\ & \mathrm{N} - \mathrm{sent} > s \cdot (\mathrm{N} - \mathrm{sent} - 1) && (0 \leq s \leq 1) \\
\implies\ & \mathrm{N} - \mathrm{sent} \geq s \cdot (\mathrm{N} - \mathrm{sent} - 1) + 1 \\
\implies\ & f \cdot (\mathrm{N} - \mathrm{sent}) \geq f \cdot (s \cdot (\mathrm{N} - \mathrm{sent} - 1) + 1) && (f \geq 0) \\
\implies\ & I \geq \Phi_{\texttt{brp}}(I)
\end{aligned}$$

Based on this comparison, we conclude that $\Phi_{\texttt{brp}}(I) \sqsubseteq I$ holds, and therefore I is indeed a wp-superinvariant.

D.2.3 Expected Number of Sent Packets

The loop-characteristic function of the while-loop of $\mathrm{wp}[\![\texttt{BRP}]\!](\mathrm{sent})$ is as follows:

$$\Phi_{\texttt{brp}}(X) = [\mathrm{sent} < \mathrm{N} \wedge \mathrm{success}] \cdot \mathrm{wp}[\![\texttt{body}_{\texttt{brp}}]\!](X) + [\mathrm{sent} \geq \mathrm{N} \vee \neg\mathrm{success}] \cdot \mathrm{sent}$$

Step 1: Propose candidate superinvariant I

$$I = [\mathrm{sent} < \mathrm{N} \wedge \mathrm{success}] \cdot (\mathrm{sent} + s \cdot (\mathrm{N} - \mathrm{sent})) + [\mathrm{sent} \geq \mathrm{N} \vee \neg\mathrm{success}] \cdot \mathrm{sent}$$

Step 2: Compute $\Phi_{\texttt{brp}}(I)$

$$\begin{aligned}
\mathrm{wp}[\![\texttt{body}_{\texttt{brp}}]\!](I) = {} & [\mathrm{sent} + 1 < \mathrm{N}] \cdot s \cdot (\mathrm{sent} + 1 + s \cdot (\mathrm{N} - \mathrm{sent} - 1)) \\
& + [\mathrm{sent} + 1 \geq \mathrm{N}] \cdot s \cdot (\mathrm{sent} + 1) \\
& + (1 - s) \cdot \mathrm{sent}
\end{aligned}$$

$$\begin{aligned}
\Phi_{\texttt{brp}}(I) = {} & [\mathrm{sent} + 1 < \mathrm{N} \wedge \mathrm{success}] \cdot s \cdot (\mathrm{sent} + 1 + s \cdot (\mathrm{N} - \mathrm{sent} - 1)) \\
& + [\mathrm{sent} + 1 = \mathrm{N} \wedge \mathrm{success}] \cdot s \cdot (\mathrm{sent} + 1) \\
& + [\mathrm{sent} < \mathrm{N} \wedge \mathrm{success}] \cdot (1 - s) \cdot \mathrm{sent} \\
& + [\mathrm{sent} \geq \mathrm{N} \vee \neg\mathrm{success}] \cdot \mathrm{sent} \\
= {} & [\mathrm{sent} + 1 \leq \mathrm{N} \wedge \mathrm{success}] \cdot s \cdot (\mathrm{sent} + 1 + s \cdot (\mathrm{N} - \mathrm{sent} - 1)) \\
& + [\mathrm{sent} < \mathrm{N} \wedge \mathrm{success}] \cdot (1 - s) \cdot \mathrm{sent} \\
& + [\mathrm{sent} \geq \mathrm{N} \vee \neg\mathrm{success}] \cdot \mathrm{sent} && \text{(Simplify)} \\
= {} & [\mathrm{sent} < \mathrm{N} \wedge \mathrm{success}] \cdot s \cdot (\mathrm{sent} + 1 + s \cdot (\mathrm{N} - \mathrm{sent} - 1)) \\
& + [\mathrm{sent} < \mathrm{N} \wedge \mathrm{success}] \cdot (1 - s) \cdot \mathrm{sent} \\
& + [\mathrm{sent} \geq \mathrm{N} \vee \neg\mathrm{success}] \cdot \mathrm{sent} && \text{(Simplify)} \\
= {} & [\mathrm{sent} < \mathrm{N} \wedge \mathrm{success}] \cdot \Big(s \cdot (\mathrm{sent} + 1 + s \cdot (\mathrm{N} - \mathrm{sent} - 1)) \\
& \quad + (1 - s) \cdot \mathrm{sent}\Big) \\
& + [\mathrm{sent} \geq \mathrm{N} \vee \neg\mathrm{success}] \cdot \mathrm{sent} && \text{(Distribute)}
\end{aligned}$$

... (continues on next page)

$$
\begin{aligned}
&\ldots \text{(continued from previous page)} \\
&= [\text{sent} < \text{N} \wedge \text{success}] \cdot \Big(s \cdot \text{sent} + s \cdot (1 + s \cdot (\text{N} - \text{sent} - 1)) \\
&\qquad + \text{sent} - s \cdot \text{sent} \Big) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{sent} && \text{(Distribute)} \\
&= [\text{sent} < \text{N} \wedge \text{success}] \cdot \Big(\text{sent} - s \cdot \text{sent} + s \cdot \text{sent} \\
&\qquad + s \cdot (s \cdot (\text{N} - \text{sent} - 1) + 1) \Big) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{sent} && \text{(Reorder)} \\
&= [\text{sent} < \text{N} \wedge \text{success}] \cdot \Big(\text{sent} + s \cdot (s \cdot (\text{N} - \text{sent} - 1) + 1) \Big) \\
&\quad + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{sent} && \text{(Simplify)}
\end{aligned}
$$

Step 3: Compare $\Phi_{\texttt{brp}}(I)$ and I

The simplified expression of $\Phi_{\texttt{brp}}(I)$ closely resembles the candidate invariant I:

$$
\begin{aligned}
I = {} & [\text{sent} < \text{N} \wedge \text{success}] \cdot (\text{sent} + s \cdot (\text{N} - \text{sent})) \\
& + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{sent}
\end{aligned}
$$

$$
\begin{aligned}
\Phi_{\texttt{brp}}(I) = {} & [\text{sent} < \text{N} \wedge \text{success}] \cdot \Big(\text{sent} + s \cdot (s \cdot (\text{N} - \text{sent} - 1) + 1) \Big) \\
& + [\text{sent} \geq \text{N} \vee \neg\text{success}] \cdot \text{sent}
\end{aligned}
$$

To assess the relationship between $\Phi_{\texttt{brp}}(I)$ and I, we compare the corresponding highlighted components:

$$
\begin{aligned}
& \text{N} - \text{sent} > \text{N} - \text{sent} - 1 \\
\implies\ & \text{N} - \text{sent} > s \cdot (\text{N} - \text{sent} - 1) && (0 \leq s \leq 1) \\
\implies\ & \text{N} - \text{sent} \geq s \cdot (\text{N} - \text{sent} - 1) + 1 \\
\implies\ & s \cdot (\text{N} - \text{sent}) \geq s \cdot (s \cdot (\text{N} - \text{sent} - 1) + 1) && (s \geq 0) \\
\implies\ & I \geq \Phi_{\texttt{brp}}(I)
\end{aligned}
$$

Based on this comparison, we conclude that $\Phi_{\texttt{brp}}(I) \sqsubseteq I$ holds, and therefore I is indeed a wp-superinvariant.

APPENDIX E

VERIFICATION TIME

E.1 Probability of Success

`sendPacket` **upper bound**	`sendPacket` **lower bound**	`BRP` **upper bound**	`BRP` **lower bound**	**verified?**	**time (s)**
x				✓	5.12
	x			✓	0.01
		x		timeout	
			x	✓	0.03
x	x			✓	5.16
x		x		✓	5.14
x			x	✓	5.11
	x	x		✓	0.03
	x		x	✓	0.04
		x	x	timeout	
x	x	x		✓	5.15
x	x		x	✓	5.15
x		x	x	✓	5.12
	x	x	x	✓	0.06
x	x	x	x	✓	5.21

TABLE E.1: The verification results and time when verifying the upper and lower bounds of the probability of success of `sendPacket` and `BRP`. An 'x' indicates that the procedure is included, and all verification times are an average of 5 trial runs. Note that the procedure verifying the trivial lower bound of `sendPacket` is included.

E.2 Number of Failures

`sendPacket` **upper bound**	`sendPacket` **lower bound**	`BRP` **upper bound**	`BRP` **lower bound**	**verified?**	**time (s)**
x				✓	0.20
	x			✓	0.01
		x		timeout	
			x	✓	0.01
x	x			✓	0.21
x		x		✓	0.30
x			x	✓	0.21
	x	x		✓	0.11
	x		x	✓	0.02
		x	x	timeout	
x	x	x		✓	0.31
x	x		x	✓	0.22
x		x	x	✓	0.31
	x	x	x	✓	0.12
x	x	x	x	✓	5.32

TABLE E.2: The verification results and time when verifying the upper and lower bounds of the expected number of failed transmissions of `sendPacket` and `BRP`. An 'x' indicates that the procedure is included, and all verification times are an average of 5 trial runs. Note that the procedures verifying the trivial lower bounds of both programs are included.

E.3 Number of Sent Packets

`BRP` **upper bound**	`BRP` **lower bound**	**verified?**	**time (s)**
x		✓	0.39
	x	✓	0.01
x	x	✓	0.11

TABLE E.3: The verification results and time when verifying the upper and lower bounds of the expected number of sent packets of `BRP`. An 'x' indicates that the procedure is included, and all verification times are an average of 5 trial runs. Note that the procedure verifying the trivial lower bounds is included.