BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

GARBAGE COLLECTOR FOR PNTALK OBJECTS

GARBAGE COLLECTOR OBJEKTŮ JAZYKA PNTALK

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **FILIP ŠTĚPÁN**
AUTOR PRÁCE

SUPERVISOR **Ing. RADEK KOČÍ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

Abstract

This thesis presents a comprehensive investigation into garbage collection techniques explicitly tailored for PNtalk, with a focus on using industry-standard automatic memory management. It explores the necessity, principles, and advantages of automatic memory management, highlighting the significance of garbage collection algorithms in optimizing memory usage and improving application performance. On this theoretical basis, the thesis deals with the garbage collector's design, implementation, testing, and benchmarking.

Abstrakt

Tato práce představuje komplexní výzkum technik garbage collection přizpůsobených speciálně pro PNtalk se zaměřením na použití standardní automatické správy paměti. Tato práce zkoumá nutnost, principy a výhody automatické správy paměti a zdůrazňuje význam algoritmů garbage collection pro optimalizaci využití paměti a zvýšení výkonu aplikací. Na základě těchto teoretických základů se práce zabývá návrhem, implementací, testováním a srovnávacím testováním garbage collectoru.

Keywords

garbage collection, garbage collector, memory management, automatic memory management, object oriented petri nets, petri nets, PNtalk

Klíčová slova

garbage collection, garbage collector, správa paměti, automatické správa paměti, objektově orientované petriho sítě, petriho sítě, PNtalk

Reference

ŠTĚPÁN, Filip. *Garbage collector for PNtalk objects.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Radek Kočí, Ph.D.

Rozšířený abstrakt

Garbage collection je klíčovou součástí vývoje moderního softwaru, zejména v aplikacích náročných na paměť. Tento abstrakt představuje komplexní průzkum technik GC přizpůsobených pro PNtalk s cílem zlepšit efektivitu správy paměti, výkon a škálovatelnost. Práce začíná zkoumáním nutnosti a principů automatické správy paměti a zdůrazňuje výhody algoritmů GC pro optimalizaci využití paměti a snižování režie manuální správy paměti.Různé algoritmy GC, včetně algoritmů mark-sweep, mark-compact, generational a reference counting, jsou podrobně zkoumány z hlediska jejich vhodnosti a použitelnosti pro PNtalk. Na základě těchto teoretických základů se práce zabývá návrhem, implementací, testováním a srovnávacím testováním garbage collectoru optimalizovaného speciálně pro PNtalk. Fáze návrhu objasňuje datové struktury, algoritmy a optimalizační strategie zaměřené na maximalizaci efektivity, spolehlivosti a přizpůsobivosti. Výsupem je garbage collector který se snadno integruje do systému PNtalk. Důsledné testovací metodiky využívající JUnit Jupiter pro testování výkonu a škálovatelnosti, zátěžové testování ověřují správnost, robustnost a výkonnostní charakteristiky garbage collectoru při různých pracovních zátěžích. Testování výkonu a srovnávací testy poskytují důkazy o efektivitě, škálovatelnosti a využití prostředků garbage collectoru, které jsou podkladem pro optimalizaci a vodítkem pro rozhodování o nasazení v reálném prostředí. Práce nejen zvyšuje výkonnost, škálovatelnost a spolehlivost systému PNtalk, ale také pokládá základy pro další zkoumání a optimalizace.

Garbage collector for PNtalk objects

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Radek Kočí. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Filip Štěpán
May 13, 2024

Acknowledgements

I would like to thank Mr. Kočí for his help and guidance.

Contents

List of Figures

Chapter 1

Introduction

In the realm of computer science and software engineering, the efficient management of memory allocation and deallocation is paramount for the optimal performance of any software system. This becomes particularly crucial in environments where resources are limited or where the system's responsiveness directly impacts its functionality. Object-Oriented Petri Nets represent a powerful paradigm for modeling concurrent systems, offering a structured approach to representing both state and behavior. However, the dynamic nature of OOPNs poses unique challenges in memory management, especially in long-running simulations where objects are constantly created and destroyed.

This thesis explores the design and implementation of a garbage collector tailored for PNtalk which is a specific implementation of OOPN. Garbage collection, a fundamental technique in modern programming languages, automates the process of reclaiming memory occupied by objects that are no longer in use. By implementing a garbage collector within the context of PNtalk simulations, we aim to address the inherent complexities of memory management, thereby enhancing the performance and scalability of PNtalk.

1.1 Goals

The objectives of this thesis are twofold: firstly, to analyze the memory management challenges inherent in OOPN simulation environments, and secondly to propose and implement an efficient garbage collector tailored to address these challenges. To achieve these objectives, we will examine the existing literature on garbage collection techniques, with a focus on those applicable to dynamic and object-oriented systems. Subsequently, we will delve into the intricacies of OOPNs, identifying patterns and usage that impact memory consumption and fragmentation.

The proposed garbage collector will be designed to integrate seamlessly with existing PNtalk implementation, minimizing overhead while maximizing memory reclamation. We will evaluate the effectiveness of the garbage collector through rigorous testing and performance analysis, comparing its impact on memory utilization, execution time, and overall system responsiveness against baseline implementations without garbage collection.

By mitigating the memory overhead associated with no memory management, the proposed garbage collector has the potential to enhance the scalability and reliability of PNtalk.

1.2 Structure

This thesis is split into 7 different chapters, including this one. The second chapter provides a brief overview of manual memory management and comprehensive overview of automatic memory management techniques. It explores various garbage collection algorithms, such as mark-sweep, mark-compact, generational, and reference counting, discussing their strengths, weaknesses, and suitability for different application scenarios. The third chapter looks at the OOPNs and PNtalk, it discusses the various components of PNtalk and it's structure. The fourth chapter builds upon the theoretical underpinnings established in the previous chapter, this section delves into the conceptualization and design of a garbage collector tailored for PNtalk. The fifth chapter shifts focus to the practical realization of the garbage collector, detailing the implementation process from codebase setup to integration with PNtalk. The sixth chapter, examines the testing, validation, and performance evaluation of the garbage collector in various test cases. Finally the last chapter summarizes key findings of this thesis, and evaluates the implemented garbage collector, it also looks at possible future improvements that could be implemented.

Chapter 2

Memory management

This chapter provides an in-depth examination of various memory management techniques, including manual memory management and automatic memory management. We will take a closer look at garbage collection algorithms such as mark-sweep, copying GC, generational GC and more. The information for this chapter was taken from [4] and [1]

2.1 Manual memory management

Manual memory allocation and deallocation are fundamental concepts in programming where the programmer explicitly manages the allocation and release of memory resources within a software program. This process involves requesting memory from the system when needed, and explicitly releasing it when it is no longer required. In languages such as C and C++, manual memory management is a common practice, giving developers fine-grained control over memory usage but also requiring them to handle memory-related tasks manually.

2.1.1 Memory Allocation

When a program needs memory to store data, it requests a block of memory from the operating system. In manual memory allocation, this is typically done using functions like malloc() in C or new operator in C++. The requested memory block is then reserved for the program's use, and a pointer to this memory location is returned. The programmer is responsible for ensuring that enough memory is allocated to store the data required by the program, taking into account the type and size of the data being stored.

2.1.2 Memory Deallocation

Once the allocated memory is no longer needed, it should be returned to the system to prevent memory leaks and conserve system resources. In manual memory management, deallocation is explicitly performed by the programmer using functions like free() in C or delete operator in C++. The programmer must ensure that all dynamically allocated memory is properly deallocated when it is no longer needed. Failure to deallocate memory can result in memory leaks, where memory that is no longer in use remains allocated, leading to inefficient memory usage and potential system instability over time.

2.1.3 Advantages and disadvantages

Manual memory management provides developers with fine-grained control over memory allocation and deallocation, allowing for performance optimization and predictable memory behavior. However, it introduces complexity and error-prone situations such as memory leaks or dangling pointers, which can be challenging to debug and resolve. Additionally, manual memory management requires developers to invest extra effort into resource management tasks, increasing development time and maintenance overhead. Despite its advantages in performance optimization and control, the lack of safety guarantees and the potential for errors make manual memory management a less desirable approach, particularly in complex software systems where reliability and maintainability are paramount.

2.2 Automatic memory management

Automatic memory management, also known as garbage collection, is a programming technique that automates the process of memory allocation and deallocation, relieving developers from the burden of manually managing memory resources. In contrast to manual memory management where developers explicitly allocate and deallocate memory, automatic memory management employs algorithms and mechanisms to identify and reclaim memory that is no longer in use, typically referred to as garbage. Garbage collection techniques vary, but they generally involve periodically scanning the program's memory space to identify objects that are no longer reachable or referenced by the program. Once identified, these unreferenced objects are reclaimed, freeing up memory for future use. Automatic memory management is widely used in modern programming languages such as Java, C#, and Python, offering several benefits including improved developer productivity, reduced risk of memory-related errors, and enhanced application reliability. However, it also introduces overhead in terms of CPU and memory usage, and the periodic pauses associated with garbage collection cycles may impact application performance, particularly in real-time or latency-sensitive systems. Despite these challenges, automatic memory management remains a powerful tool for simplifying memory management tasks and improving the robustness of software systems, especially in environments where manual memory management would be impractical or error-prone.

2.3 Garbage collection

Garbage collection (GC) techniques vary in their approach to identifying and reclaiming memory that is no longer in use by the program. Different types of garbage collection algorithms have been developed over the years, each with its own advantages, disadvantages, and suitability for various specific use cases.

2.3.1 Performance metrics

Comparing garbage collector techniques involves evaluating various metrics that assess their performance, efficiency, and impact on system behavior. These metrics provide valuable insights into the strengths, weaknesses, and trade-offs of different GC techniques

Memory overhead

Memory overhead refers to the additional memory consumed by a garbage collector beyond the memory required to store the application's data and code. It encompasses various components, including data structures, bookkeeping overhead, and auxiliary memory, utilized by the GC to manage memory allocation and deallocation. A lower memory overhead is desirable as it minimizes the impact on the application's overall memory usage and improves resource efficiency. High memory overhead can lead to increased memory consumption, reduced available memory for application data, and potential performance degradation. Therefore, optimizing memory overhead is crucial for ensuring efficient memory management and maximizing the scalability and performance of applications.

Throughput

Throughput refers to the rate at which an application performs useful work between successive GC cycles. It is a measure of the application's overall execution speed and efficiency. Higher throughput indicates faster execution and better utilization of system resources. Garbage collection techniques that optimize throughput aim to minimize the time spent on GC activities relative to the time spent executing application code. Achieving high throughput is particularly important in high-performance computing environments, web servers, and other latency-sensitive applications.

Pause times

Pause times refer to the periods during which application execution is suspended or paused while garbage collection activities are performed. Minimizing pause times is crucial for maintaining application responsiveness, especially in interactive or real-time systems where user experience is paramount. Long pause times can lead to delays in user interaction, reduced throughput, and degraded application performance. Garbage collection techniques that aim to reduce pause times, such as concurrent or incremental GC, allow application threads to execute concurrently with GC activities, thereby minimizing the impact on application responsiveness.

2.3.2 Garbage collection roots

Garbage collection roots are objects or memory locations that are directly accessible or known to the runtime environment and are therefore considered as starting points for the GC traversal process. These roots typically include global variables, static variables, CPU registers, and local variables in active threads' call stacks. GC roots serve as references from which the garbage collector can traverse the object graph to identify and mark reachable objects. Objects that are not reachable from any GC root are considered unreachable and eligible for garbage collection. Ensuring that all active references are appropriately managed as GC roots is essential for maintaining memory integrity and preventing premature deallocation of live objects. By accurately identifying and maintaining GC roots, the garbage collector can effectively manage memory resources and reclaim unused memory, contributing to efficient memory utilization and improved application performance.

2.3.3 Reference counting

Reference counting is a straightforward garbage collection technique that tracks the number of references to each object in memory. The basic premise is to associate a reference count with each object, indicating how many references point to it. When an object's reference count drops to zero, it signifies that the object is no longer reachable from the program's execution context and can be safely deallocated.

Advantages

- Immediate Reclamation: Reference counting immediately reclaims memory when an object's reference count drops to zero, allowing for prompt resource cleanup. This proactive approach can prevent memory leaks and improve overall memory utilization.
- Low Overhead: Reference counting typically incurs low runtime overhead compared to other garbage collection techniques, such as mark-sweep or copying garbage collection. This makes it suitable for applications with strict performance requirements or limited computational resources.

Disadvantages

- Inefficient for Cyclic References: Reference counting is inefficient at handling cyclic references, where objects reference each other in a loop. Even if a cyclic group of objects is collectively unreachable from the program's execution context, their reference counts never reach zero due to their mutual references, leading to memory leaks.
- Overhead of Reference Count Updates: Incrementing and decrementing reference counts for each object reference operation can introduce overhead, particularly in multi-threaded environments where atomic operations may be required to ensure thread safety.
- Difficulty in Handling Weak References: Reference counting does not inherently support weak references, which are references that do not prevent the referenced object from being deallocated. Implementing weak references with reference counting requires additional mechanisms, potentially complicating the garbage collection process.

2.3.4 Mark-Sweep

Mark-Sweep is a classic garbage collection algorithm that operates in two phases: marking and sweeping. It is designed to reclaim memory occupied by unreachable objects by traversing the entire object graph and identifying objects that are still reachable from the program's execution context.

Mark phase

- Identification of Reachable Objects: The garbage collector starts from a set of known root objects, such as global variables, local variables, and stack frames, which are guaranteed to be reachable. It traverses the object graph recursively, marking each encountered object as reachable.

- Tracing Algorithm: Marking is typically performed using a tracing algorithm, such as depth-first search (DFS) or breadth-first search (BFS), which systematically explores the object graph starting from the root objects.

Sweep phase

- Reclamation of Unreachable Objects: Once all reachable objects have been marked, the garbage collector sweeps through the entire heap, deallocating memory for objects that were not marked as reachable during the marking phase.
- Memory Reclamation: Unreachable objects are identified by the absence of a mark. The memory occupied by these objects is reclaimed and made available for future allocations.

Advantages

- Efficient Handling of Cyclic References: Mark-Sweep is effective at handling cyclic references, where objects reference each other in a loop, by traversing the entire object graph and marking reachable objects. As long as at least one object in a cyclic group is reachable, the entire group will be retained.
- Immediate Reclamation: Once unreachable objects have been identified during the sweeping phase, their memory is immediately reclaimed, ensuring efficient memory usage and minimizing the risk of memory leaks.
- Simplicity: Mark-Sweep is relatively straightforward to implement and understand, making it a popular choice for garbage collection in many runtime environments and programming languages.

Disadvantages

- Pause Times: The mark-sweep algorithm typically requires a stop-the-world pause during the marking and sweeping phases, where all program execution is halted while garbage collection is performed. These pause times can be disruptive in interactive or real-time systems, affecting application responsiveness.
- Traversal Overhead: Traversing the entire object graph during the marking phase can introduce significant overhead, particularly in applications with large heaps or complex object graphs. This overhead may impact application performance and scalability.

2.3.5 Mark-Compact

Mark-Compact is a type of garbage collection algorithm that combines the marking and sweeping phases of the mark-sweep algorithm with an additional compaction step. The primary goal of mark-compact garbage collection is to reclaim memory occupied by unreachable objects while also compacting memory to reduce fragmentation and optimize memory usage.

Mark/sweep phase

Same as mark-sweep algorithm, except in some implementations an additional information about each reachable object's new location after compaction is stored.

Compact phase

- Once memory has been reclaimed, the mark-compact algorithm compacts the remaining live objects to eliminate fragmentation and optimize memory usage.
- Live objects are moved to consecutive memory locations, compacting the heap and reducing the amount of wasted space caused by fragmentation.
- Object references are updated to reflect the new memory locations of relocated objects, ensuring that object relationships remain intact.

Advantages

- Fragmentation Reduction: Mark-Compact garbage collection effectively reduces memory fragmentation by compacting live objects into contiguous memory regions. This reduces wasted space and optimizes memory usage, improving overall memory efficiency.
- Same advantages as mark-sweep

Disadvantages

- Increased Complexity: The addition of the compaction phase adds complexity to the garbage collection process, requiring additional bookkeeping and memory movement operations. This increased complexity may introduce overhead and impact overall garbage collection performance.
- Potentially Longer Pause Times: The compaction phase of mark-compact garbage collection may increase pause times compared to mark-sweep, as it involves additional memory movement operations. Longer pause times can impact application responsiveness, particularly in real-time or interactive systems.
- Additional Memory Overhead: Compacting live objects into contiguous memory regions may require additional memory to temporarily store objects during compaction. This can increase memory overhead, particularly in systems with limited memory resources.

2.3.6 Generational Garbage Collection

Generational garbage collection is a technique that leverages the observation that most objects become garbage shortly after they are allocated. This technique divides objects into different generations based on their age, typically distinguishing between young and old generations. Generational garbage collection focuses garbage collection efforts on younger generations, as they tend to contain a higher proportion of short-lived objects, while older generations are collected less frequently.

Generational Divisions

- Objects are initially allocated in the young generation, often referred to as the nursery or nursery space. The young generation is typically smaller in size and optimized for rapid allocation and collection.
- As objects survive garbage collection cycles in the young generation, they are promoted to older generations. Objects that survive multiple garbage collection cycles in the older generations are considered long-lived and are eventually collected during a full garbage collection cycle.

Minor Garbage Collection

- Minor garbage collection, also known as young generation garbage collection, focuses on reclaiming memory in the young generation.
- During a minor garbage collection cycle, the garbage collector identifies and collects garbage objects in the young generation using techniques such as copying or marking-sweeping. Surviving objects are then promoted to the next older generation.

Major Garbage Collection

- Major garbage collection, also known as full garbage collection or global garbage collection, targets the entire heap, including both young and old generations.
- Major garbage collection typically occurs less frequently than minor garbage collection and involves reclaiming memory across all generations. This process may be more complex and time-consuming than minor garbage collection due to the larger heap size and the need to traverse objects in multiple generations.

Advantages

- Efficient Handling of Short-Lived Objects: By focusing garbage collection efforts on the young generation, generational garbage collection can efficiently reclaim memory occupied by short-lived objects, reducing the overhead of full garbage collection cycles.
- Reduced Pause Times: Minor garbage collection cycles, which target the young generation, can be completed more quickly than full garbage collection cycles. This can result in shorter pause times and improved application responsiveness.
- Adaptive Performance: Generational garbage collection adapts to the allocation and usage patterns of the application over time. By promoting surviving objects to older generations, the garbage collector can prioritize garbage collection efforts where they are most needed.

Disadvantages

- Potential for Premature Promotion: Objects that survive multiple minor garbage collection cycles may be prematurely promoted to older generations, leading to increased memory usage and longer garbage collection times in the long term.

- Increased Complexity: Generational garbage collection introduces additional complexity to the garbage collection process, including managing multiple generations, deciding when to promote objects between generations, and coordinating garbage collection cycles across generations.
- Tuning Overhead: Configuring and tuning generational garbage collection parameters, such as generation sizes and promotion thresholds, may require careful analysis and experimentation to achieve optimal performance for specific applications and workloads.

2.3.7 Copying Garbage Collection

Copying garbage collection is a memory management technique that divides the heap into two semi-spaces and performs garbage collection by copying live objects from one semi-space to the other. This technique is particularly effective for reclaiming memory occupied by short-lived objects and reducing fragmentation.

Heap Division

- The heap is divided into two semi-spaces, often referred to as the from-space and the to-space. Initially, all object allocations occur in the from-space.
- The to-space is initially empty and serves as the destination for copied live objects during garbage collection.

Minor Garbage Collection

- During a minor garbage collection cycle, the garbage collector scans the from-space to identify live objects.
- Live objects are copied from the from-space to the to-space, leaving behind only garbage in the from-space.
- Object references in the copied objects are updated to point to their new locations in the to-space.
- Once all live objects have been copied, the roles of the from-space and to-space are swapped, making the to-space the new from-space for subsequent allocations.

Major Garbage Collection

- Major garbage collection, also known as full garbage collection, may be performed when the to-space becomes full or when the from-space is significantly fragmented.
- During a major garbage collection cycle, all live objects are copied from the from-space to the to-space, compacting memory and reclaiming space occupied by garbage.
- Unlike minor garbage collection, major garbage collection involves copying all live objects in the heap, not just those in the from-space.

Advantages

- Fragmentation Reduction: Copying garbage collection effectively reduces memory fragmentation by compacting live objects into contiguous memory regions in the to-space. This reduces wasted space and optimizes memory usage, leading to improved memory efficiency.
- Immediate Reclamation: Garbage collection occurs incrementally during minor garbage collection cycles, ensuring that memory occupied by unreachable objects is immediately reclaimed and made available for reuse.
- Simplicity: Copying garbage collection is relatively simple to implement and understand, making it a popular choice for memory management in many runtime environments and programming languages.

Disadvantages

- Memory Overhead: Copying live objects from one semispace to another during garbage collection can introduce memory overhead, particularly if the to-space is not sufficiently large to accommodate all live objects from the from-space.
- Potentially Longer Pause Times: Minor garbage collection cycles may introduce pause times during which application execution is suspended while live objects are copied from the from-space to the to-space. These pause times can impact application responsiveness, particularly in real-time or interactive systems.
- Increased Copying Costs: Copying all live objects during major garbage collection cycles can introduce additional copying costs, particularly in systems with large heap sizes or complex object graphs. These copying costs may impact garbage collection performance and scalability.

2.3.8 Concurrent/parallel garbage collection

Concurrent garbage collection is an approach to reclaiming memory in managed runtime environments that aims to minimize pause times and maintain application responsiveness by performing GC activities concurrently with the execution of application threads. Unlike traditional stop-the-world (STW) GC, where application execution is halted during GC cycles, concurrent GC techniques allow the application to continue executing while GC activities occur in parallel.

Key components

The key components of concurrent garbage collection include concurrent marking, sweeping, and optionally, compaction. Concurrent marking involves traversing the object graph to identify reachable objects while allowing application threads to execute concurrently. Concurrent sweeping deallocates memory for unreachable objects without halting application threads, ensuring continuous execution. Optional concurrent compaction reorganizes memory to reduce fragmentation, improving memory locality and efficiency. These components work together to minimize pause times and maintain application responsiveness, enabling smooth user experiences in managed runtime environments.

Challenges

Implementing concurrent garbage collection poses several challenges due to the inherent complexity of managing memory concurrently with the execution of application threads. One significant challenge is coordinating GC activities with ongoing application execution to ensure correctness and consistency while minimizing pause times. This requires implementing sophisticated synchronization mechanisms and concurrency control techniques to manage access to shared data structures and resources. Additionally, dealing with concurrent access to mutable objects and managing inter-thread communication introduces the risk of race conditions and synchronization errors, necessitating careful design and testing. Furthermore, optimizing the performance of concurrent GC requires balancing the trade-offs between throughput, pause times, and resource utilization, which may vary depending on the characteristics of the application workload and the underlying hardware architecture. Overall, implementing concurrent GC requires expertise in concurrent programming, memory management, and system optimization, as well as a thorough understanding of the specific requirements and constraints of the targeted runtime environment.

Chapter 3

PNtalk

In this chapter, we take a closer look at PNtalk, exploring its design principles, features and capabilities. By understanding the underlying architecture and functionality of PNtalk, this will allow us to make more informed decisions when it comes to implementing a garbage collector. We will examine all of the key components for this thesis, such as its syntax, semantics and more. We will focus only on the current implementation in Java, and since at the time of writing this it is not finished, some of the information in this chapter might be outdated. The information for this chapter was taken from [2] and [3].

PNtalk is a language and a system based on Object-Oriented Petri Nets(OOPNs). PNtalk language is a specific implementation of OOPN, PNtalk also specifies some facts in which the OOPN definition leaves some latitude.

3.1 Object-Oriented Petri Nets

Object-Oriented Petri represent an extension of traditional Petri nets that incorporates object-oriented concepts from software engineering. OOPNs provide a powerful modeling framework for describing concurrent and distributed systems in a modular, hierarchical, and reusable manner. By combining the formalism of Petri Nets with the principles of object-oriented programming, OOPNs offer a flexible and intuitive approach to modeling complex systems, ranging from communication protocols and workflow systems to manufacturing processes and software architectures.

3.1.1 Key concepts

1. Places and Transitions: Places in OOPNs represent system states or conditions. They serve as containers for tokens, which signify the presence of entities or resources within the system. Transitions represent events or actions that can occur in the system. They serve as triggers for state changes and token movements between places.

2. Objects and classes: In OOPNs objects encapsulate state and behavior. Objects represent tangible entities within the system and interact with each other through sending messages. Classes serve as blueprints for creating and managing objects in the system.

3. Inheritance and Polymorphism: OOPNs support inheritance and polymorphism, enabling the modeling of hierarchical structures and behavioral variations within the

system. Inheritance allows classes to inherit properties and behaviors from parent classes, promoting code reuse and modularity. Polymorphism allows objects of different classes to be treated interchangeably, providing flexibility and extensibility in system design.

3.2 Messaging

Sending a message can only be triggered as an action during transition. Messages consist of selectors and optional arguments. There are three different type of messages:

1. Unary messages: Messages that are sent to an object without any other information. For example, `C1 new` is a unary message.

2. Binary messages: Messages consisting of operators (often arithmetic). They are binary because they always involve only two objects: the receiver and the argument object. For example in `10 + 20`, + is a selector sent to the receiver 10 with argument 20.

3. Keyword messages are messages consisting of one or more keywords, each ending with a colon (:) and taking an argument. For example in `o doit: 1`, o is the receiver and the selector doit: takes the argument 1. Objects can also send messages to themselves by setting the receiver to the keyword self.

3.3 Places, transitions and edges

Every place in OOPN has a name associated with it, and can also have set the default token contained within it. Places can optionally have initial action associated with them. PNtalk allows for the default token to be a variable, but the variable in question has to be set to a value by the associated initial action.

Places and transition are connected by edges. Edges have edge expression associated with them, here is an example of an edge expression:

```
2‘#e
```

In this expression `#e` represents a symbol and the number 2 represents the amount of symbols required. There are three different types of edges:

1. Input edge: These edges connect places to transitions, indicating that the transition requires tokens from those places to fire. Input edges represent the prerequisites or conditions necessary for a transition to occur, ensuring that the system behaves according to predefined rules or constraints.

2. Output edge: When a transition with output edges fires, it generates tokens in its output places, reflecting the outcomes or effects of the transition's execution.

3. Test edge: Testing edges are a specialized type of input edge that represents a conditional dependency between a transition and a place. Tests whether the required tokens exists in the connected place.

As mentioned before transitions are the only way to send a message in the java implementation of PNtalk. This means that the only way to create new objects is through transitions, these objects can also be assigned to a variable and then stored inside places.

3.4 Nets

Places and transitions connected by edges form nets. Every object in PNtalk can contain two types of nets:

1. Object net: These represent all of the objects attributes and its activity
2. Method net: These specify the reaction of an object to a method call (message from another object). Every method network has a message template associated with it. The creation of method networks is dynamic, and upon the method finishing the net is destroyed. Method nets also contain two „special“ kind of places. Parameter places where the arguments are stored and the return place that is used to return the result to the caller.

Below 3.1 we can see an example of both object net and method net.

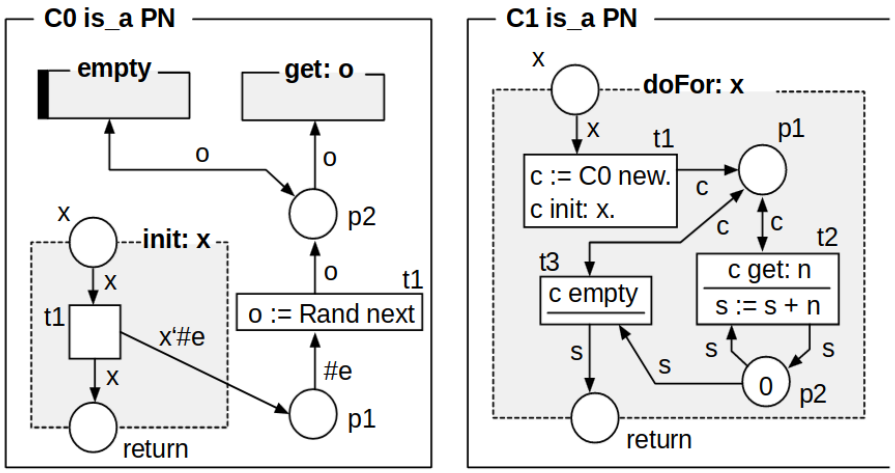

Figure 3.1: Example of OOPN (taken from [3]).

3.5 Structure of the current PNtalk implementation

The description of the structure is only gonna be partial as it is not necessary to understand all of PNtalk for this thesis. For the graphical represantion of the structure see 3.2. Few notable things about the structure are that PNPlaces contain PNPMultiSettokens and these tokens contain a hashmap with PNProxy and an integer, the integer represents multiplicity and PNProxy a value, which can be either string, symbol, number, object or a compiled class. Every transition has transition goals these represent preconditions (input edges) and postconditions (output edges) of a transition.

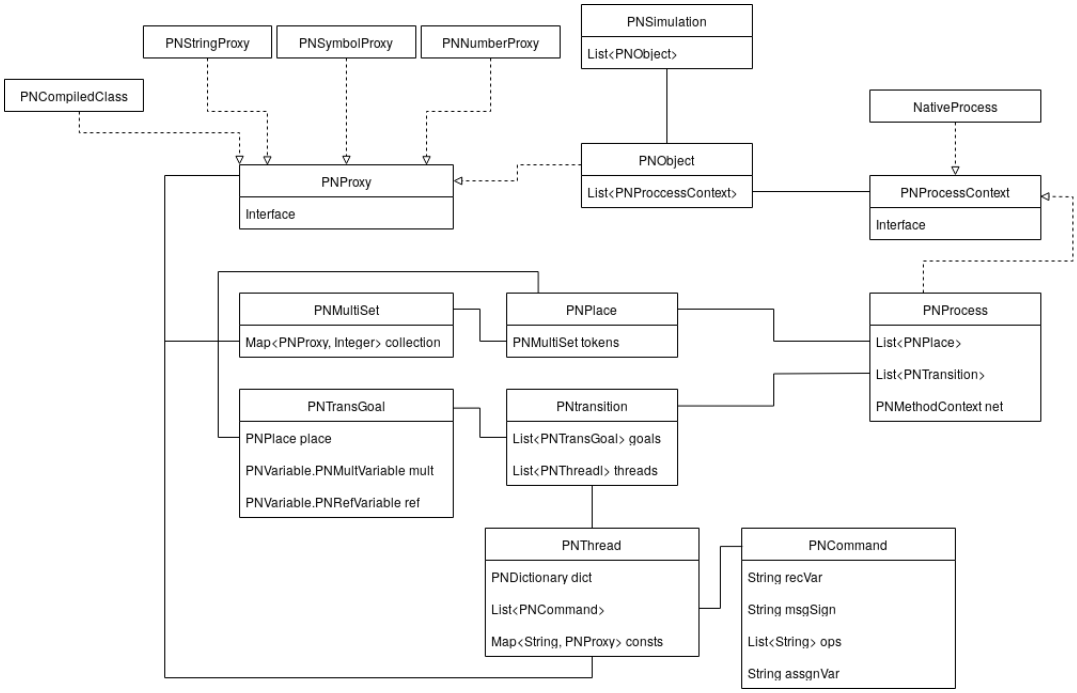

Figure 3.2: PNtalk structure (simplified).

3.5.1 PNSimulation

Represents a simulation, it contains PNObject list, PNCalendar and PNTime. PNCalendar is used for schedulling events, these events are executed in a case that there are no transition to be fired for any of the PNObject in the simulation's list. PNTime is modified when even is executed. The list of PNObjects represents all the existing objects in the simulation.

3.5.2 PNObject

PNObject is a representation of an instantiated OOPN class, it contains PNCompiled-Class (class which the object was instantiated from), PNProcess which represents an object net and finally list of PNProcessContext. PNProcessContext represents the context of a process, the context can either be a method net or a native method.

3.5.3 PNBinding

PNBinding contains all the necessary information needed to fire and successfully finish a transition. Before a PNThread can be created it firts has to be bound successfully.

3.5.4 PNThread

PNThreads represent a progress of a transition and also the current state of the transition. It has three possible states and those are RUNNABLE, WAITINGOP and FINISHINGOP.

- RUNNABLE - in this state the thread can perform operations, this is the default state of a thread after it has been instantiated.
- WAITINGOP - waiting for an operation to finish, the thread can get into this state when it calls a method.

- FINISHINGOP - Informs the thread of the fact that the process it was waiting for has finished.

Chapter 4

Garbage Collector design

This chapter delves into the conceptualization, design decision and architecture of a memory management system tailored for PNtalk. It is worth mentioning that since the PNtalk implementation is done in java, and java does not support any manual memory management, that my garbage collector will not be implemented in the traditional way. Pntalk works on a basis of a virtual machine this means that PNSimulation holds all PNObjects in memory and never removes them (the sole exception being method nets, that get removed upon finishing). So what my garbage collector will need to do are these following things:

1. Determine which PNObjects in memory will be considered GC roots.
2. Find all references that roots contain.
3. Traverse these reference recursively and mark them as reachable.
4. Traverse the entire memory of PNSimulation and remove all unreachable objects.

4.1 Goals

One of the main goals of the garbage collector design is to minimize changes to the existing PNtalk implementation and encapsulate memory management functionality within the garbage collector itself. By containing all memory management operations within the garbage collector, we aim to remove the risk of creating changes that are at odds with the future development of PNtalk. Furthermore we want to put lot of emphasis on ease of testing and tunability. Tunability is a crucial aspect that enhances the garbage collector's versatility and adaptability to many different kinds of Object-Oriented Petri Nets.

4.2 Algorithms

After careful consideration I have decided to implement multiple different GC algorithms, that will be available to use for garbage collection. Different garbage collection algorithms excel under different conditions and usage patterns. By implementing multiple algorithms, we can tailor the choice of garbage collection strategy to match specific needs, such as object longevity and memory usage patterns. This allows for optimization across a wide range of scenarios, maximizing performance and efficiency. Having multiple algorithms enables direct comparison of their performance metrics. By evaluating algorithms side

by side, we will gain insights into their relative strengths and weaknesses. The specific algorithms I have chosen are sweep-mark, generational GC and on top of that parallelized mark-sweep.

4.3 Garbage collection frequency

I decided to implement garbage collection triggers only after a certain number of steps in the simulation to strike a balance between memory management and computational efficiency. By delaying garbage collection until after a predefined number of steps, I aim to minimize the frequency of GC pauses while ensuring that memory usage remains within acceptable bounds. Performing GC too frequently can introduce unnecessary overhead and potentially disrupt the simulation's flow, leading to increased pause times and reduced throughput. However, delaying GC indefinitely can result in excessive memory consumption and potential memory exhaustion, negatively impacting application performance. Therefore, by scheduling GC triggers at regular intervals based on the simulation's progress, I can optimize memory utilization while maintaining smooth and responsive simulation execution.

4.4 Identifying garbage collection roots

Identifying garbage collection roots in PNtalk simulation is difficult, because the simulator keeps all of the PNobjects in one list and we have no additional information about these objects. In a typical GC, garbage collection roots would include global variables, static variables, and local variables, nothing like that exists in the PNtalk simulation. So the main criteria I have decided to use to identify GC roots is whether the object was created before the simulation started.

4.5 Deleting garbage collection roots

The problem with deleting roots is that it is hard to determine when a root should be deleted. At first I wanted to delete roots based on their activity, meaning if there were any PNThread (3.5) created or existing the root would not get deleted, however this proved to be a mistake because the PNSimulation allows only a single transition to be executed per step. This means if I had more than one roots ready to execute their respective steps only one of them would be executed and the other one would get deleted. However there is a way to accurately determine whether or not an object should be deleted. By checking if each transition inside an object can be bound, we determine that this object can fire transitions in the future. This means we can delete a root if no transitions can be bound and if it also has no threads.

4.6 Parallelization

4.6.1 Parallel sweep

Parallel sweep allow us to reclaim memory concurrently across multiple threads. This is most likely the easiest phase to parallelize, as the task itself can easily be decomposed into smaller independent tasks, and the individual task will not need access to the same parts

of memory. Firstly, the heap (in our case the list of PNObject inside PNSimulation) must be partitioned into smaller segments, with each segment assigned to a specific thread or processor for sweeping. To achieve this I decided to include a parameter that will limit the maximum number of threads. The objects will then be evenly split between the maximum number of threads.

4.6.2 Parallel Marking

For the most part parallel marking is very similar to parallel sweeping, with the added problem of threads potentially accessing the same object. This problem can be solved quite easily and that is by using the boolean used for „marking“ the object. However for this to work I have to prevent different threads accessing the boolean at the same time, for this I used Java's synchronized keyword that allows methods to be thread-safe by ensuring that only one thread can execute it at a time. This solves all of our potential issues because if two threads access the same object, then one thread will mark and the other one will move on. It will also utilize the same concept of limiting the maximum amount of threads and splitting the work equally between them.

4.7 Garbage collector test class

I decided to create a subclass that inherits from the garbage collector class to facilitate testing and customization within the garbage collection framework. By extending the base garbage collector class, I gain the ability to add additional functionality specific to testing purposes while leveraging the existing implementation of the garbage collector for core memory management tasks. This approach allows me to isolate and test specific components or behaviors of the garbage collector in a controlled environment without modifying the original implementation. Additionally, subclassing enables me to override or extend existing methods in the garbage collector class to introduce custom behavior tailored to testing requirements.

Chapter 5

Implementation

In this chapter we will delve into the practical realization of the garbage collector, detailing the architectural components, data structures, algorithms, and implementation methodologies employed to bring the design to fruition.

5.1 Class hierarchy

GarbageCollector is the base class for all the other garbage collector implementations. Each of the garbage collector implementations has its own testing class that inherits from their respective implementations.

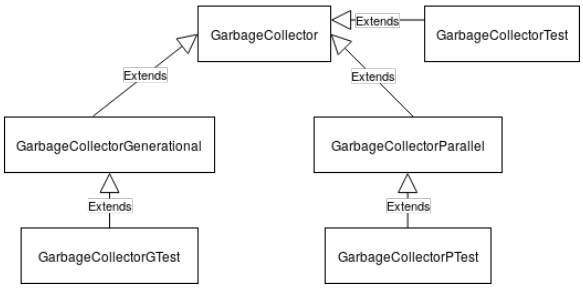

Figure 5.1: Class hierarchy diagram.

5.2 Changes to PNtalk

Before I could even start implementing the garbage collector I first had to make some changes to PNtalk itself. First of all I had to add reachable boolean to class PNObject, this was done so the garbage collector can mark PNObject. Furthermore I had to add a couple of getters:

- PNObject: added a method getComponents that returns a list of PNProcessContext.
- PNProcess: added a method getPlaces that returns a list of PNPlace, also added a method getTransitions that returns a list of PNTransition.
- PNTransition: added a method getThreads that returns a list of PNThread.

- PNPlace: added a method getTokens that returns a PNMultiSet.
- PNMultiSet: added a method getCollection that returns a that returns a Map where PNProxy is used as a key and integer is used as a value.
- PNThread: added a method getDictionary that returns PNDictionary.
- PNDictionary: added a method getContent that returns a that returns a Map where PNProxy is used as a value and string is used as a key.

These changes were made in order to be able to loop through the list of PNObjects in PNSimulation and check for references, that can either exists in PNPlace or PNThread. For PNSimulation I just added a GarbageCollector as a member variable and during PNSimulation's step method I added a call to GarbageCollector's garbageCollect method.

5.3 Mark-sweep

The first algorithm I have decide to implement was mark-sweep, mostly because it is the easiest one to implement. Implementing it allowed me more of an insight into PNtalk's structure and inner workings. This made implementing the more complex algorithms down the line much easier. Mark-sweep is implemented in GarbageCollector.java as a class. This class contains three member variables, `stepsToGc` sets the amount of steps needed to perform garbage collection, `currentStep` which serves as a step counter and list of PNObjects called `roots`, these contain all our roots, roots are inserted into the list manually before simulation.

```
public void garbageCollect(List<PNObject> components) {
    if(currentStep != stepsToGc - 1){
        currentStep += 1;
        return;
    }
    mark();
    sweep(components);

    currentStep = 0;
}
```

Listing 5.1: garbageCollect method

When PNSimulation finishes one step it calls garbageCollect 5.1 which first performs a check whether the required amount of steps has passed. If the check is failed the function does nothing and returns, otherwise it performs both mark and sweep phases and at the end sets the step counter to zero.

The mark method 5.2 starts out by looping through the list of roots and calling checkObjRefs 5.2 on each root. The first thing this function does is sets the objects reachability to true, after which it loops through a list of PNProcess and from PNProcess it acquires a list of places and transition which are used as arguments for functions checkPlaceRefs and checkThreadRefs. These functions look for a reference to a PNObject if they find it they recursively call checkObjRefs. This basically works like a depth first search except we go through the entire list to the end.

```
void mark() {
    for (PNObject obj : roots) {
        checkObjRefs(obj, true);
    }
}

void checkObjRefs(PNObject obj){
    obj.reachable = true;
    for (PNProcessContext context : obj.getComponents()) {
        if (context instanceof PNProcess) {
            PNProcess process = (PNProcess) context;

            for(PNPlace place : process.getPlaces()){
                checkPlaceRefs(place);
            }

            for(PNTransition trans : process.getTransitions()){
                for(PNThread thread : trans.getThreads()){
                    checkThreadRefs(thread);
                }
            }
        }
    }
}
```

Listing 5.2: mark and checkObjRefs methods.

The sweeping phase 5.3 is very simple we just loop through all the objects in the components list and check if the reachable boolean is set to true or false, based on that we either remove the object or set reachable to false to prepare it for the next garbage collection.

```
    void sweep(List<PNObject> components) {
        Iterator<PNObject> iterator = components.iterator();
        while (iterator.hasNext()) {
            PNObject obj = iterator.next();
            if (!obj.reachable) {
                iterator.remove();
            } else {

                obj.reachable = false;
            }
        }
    }
```

Listing 5.3: sweep method.

5.4 Generational Garbage collection

The generational garbage collector is implemented as a class GarbageCollectorGenerational and it is a child class of class GarbageCollector. This means that it has all the same methods and member variables. On top of that it has one list of PNObject (Old generation) and one list of YoungObject 5.4. YoungObjects are necessary so we can track how many cycles has a PNObject survived. Having young and old generation be part of the GarbageCollectorGenerational class has some performance drawbacks, since the objects are not directly created in the young generation, but rather added to it during marking. The class also contains two member variables, maxOldGenSize which is used as a trigger for full GC and a promotion variable that determines how many steps a YoungObject has to survive to be promoted to old generation.

```
class YoungObject{
    public PNObject obj;
    public int age;

    public YoungObject(PNObject obj){
        this.obj = obj;
        this.age = 0;
    }
}
```

Listing 5.4: YoungObject class.

The method sweepYoung 5.5 works very similarly to normal sweep except in the case that the object is not swept its age is increased, and in the case that the youngObject has reached the age for promotion to OldGeneration it is moved there and removed from the YoungGeneration list. In the case that object is unreachable and removed from the list of youngObjects it has to be also removed from the simulation's list that's passed to the method as an argument. The inherited sweep method had to be overriden so it loops over the old generation and remove objects from both the old generation and the components passed by PNSimulation.

```
void sweepYoung(List<PNObject> components) {
    Iterator<YoungObject> iterator = youngGeneration.iterator();
    while (iterator.hasNext()) {
        YoungObject yObj = iterator.next();
        PNObject obj = yObj.obj;
        if (!obj.reachable) {
            components.remove(obj);
            iterator.remove();
        } else {
            obj.reachable = false;
            if(yObj.age == promotion){
                oldGeneration.add(obj);
                iterator.remove();
            }
            else{
                yObj.age += 1;
            }
```

```
        }
    }
}
```

Listing 5.5: sweepYoung method.

The mechanism for cleaning out the older generation is very simple, 5.6 it is based on the size of our OldGeneration list, if it surpasses our set maximum size we trigger the full garbage collection. This includes sweeping both the OldGeneration and YoungGeneration.

```
@Override
public void garbageCollect(List<PNObject> components) {
    if(currentStep != stepsToGc - 1){
        currentStep += 1;
        return;
    }
    if(oldGeneration.size() <= maxOldGenSize){
        mark();
        sweepYoung();
    } else{
        mark();
        sweep(components);
        sweepYoung();
        clearRoots(components);
    }
    currentStep = 0;
}
```

Listing 5.6: garbageCollect method.

Mark method remains the same as in the parent class, but the checkObjRefs 5.7 method had to be overridden so that when we reach an object that is not root, it is addded to the YoungGeneration list.

```
@Override
void checkObjRefs(PNObject obj){
    obj.reachable = true;
    youngGeneration.add(new YoungObject(obj));

    for (PNProcessContext context : obj.getComponents()) {
        if (context instanceof PNProcess) {
            PNProcess process = (PNProcess) context;

            for(PNPlace place : process.getPlaces()){
                checkPlaceRefs(place);
            }

            for(PNTransition trans : process.getTransitions()){
                for(PNThread thread : trans.getThreads()){
                    checkThreadRefs(thread);
                }
            }
```

```
        }
    }
}
```

Listing 5.7: checkObjRefs method.

5.5 Parallel mark-sweep

The parallel mark sweep garbage collector is implemented as a class GarbageCollectorParallel and it is a child class of class GarbageCollector.

The biggest issue with parallel marking 5.8 is that it is possible for different threads to get access the same object. This is solved by adding a synchronized method to PNObject. Once a thread gets access to a block it immedietally calls the method `tryLock`, this practically locks the object and ensures other threads cant access it. For the splitting of the list of objects between threads I first divide the number of objects by the maximum amount of threads and assign it to variable objsPerThread, then every thread gets rounded down amount of objsPerThread as a sublist. If there are any remaining objects they are passed to another thread, this makes it so the maximum amount of threads is technically maxThreads + 1. Once all threads are started we wait for all of them to finish to avoid any overlaps with the sweeping phase.

```
void parallelMark() {
    List<markThread> threads = new ArrayList<>();
    float objsPerThread = roots.size()/maxThreads;
    if(objsPerThread < 1){
        mark();
        return;
    }
    else{
        int remainder = roots.size() % maxThreads;

        for (int i = 0; i < (roots.size() - remainder);
        i += Math.floor(objsPerThread)) {

            int endIndex =
            (int) Math.min(i + Math.floor(objsPerThread), roots.size());
            List<RootObject> sublist = roots.subList(i, endIndex);

            markThread thread = new markThread(sublist);
            threads.add(thread);
            thread.start();
        }
        if(remainder > 0){
            //creates new thread with the last portion of the list
        }
    }
    for (markThread thread : threads) {
        //waiting for all threads to finish
```

```
    }
}
```

Listing 5.8: parallelMark method.

Parallel sweep 5.9 has the exact same mechanism for splitting up the list of PNObjects between threads. The only things that differ are the work that the threads do and how sublist have to be handled. Since sublists are in essence views into the original list we cannot directly delete these objects as it would mess up the thread's internal iterators. The solution for this was simply setting the value of PNObject to null, these values are deleted after all threads finish.

```
void parallelSweep(List<PNObject> components) {
    List<sweepThread> threads = new ArrayList<>();
    float objsPerThread = components.size()/maxThreads;
    if(objsPerThread < 1){
        sweep(components);
        return;
    }
    else{
        int remainder = components.size() % maxThreads;

        for (int i = 0; i < (components.size() - remainder);
        i += Math.floor(objsPerThread)) {

            int endIndex =
            (int) Math.min(i + Math.floor(objsPerThread),
            components.size());
            List<PNObject> sublist = components.subList(i, endIndex);

            sweepThread thread = new sweepThread(sublist);
            threads.add(thread);
            thread.start();
        }
        if(remainder > 0){
            //creates new thread with the last portion of the list
        }
    }
    for (sweepThread thread : threads) {
        //waiting for all threads to finish
    }
    components.removeAll(Collections.singleton(null));;
}
```

Listing 5.9: parallelSweep method.

Chapter 6

Testing and Benchmarking

The testing and benchmarking chapter of this thesis is dedicated to the rigorous evaluation and validation of the developed garbage collection system for PNtalk. In this chapter, we delve into the methodologies, procedures, and results of various tests aimed at assessing the performance, reliability, and scalability of the garbage collection system under different workloads and scenarios. Through systematic testing, we aim to validate the correctness of the implementation, identify potential issues or shortcomings, and measure the system's performance against predefined tests.

JUnit Jupiter was utilized as the primary testing framework throughout the development of the garbage collector, offering a modern and feature-rich platform for writing and executing unit tests in Java, the framework was already being utilized in PNtalk prior to development of GC.

6.1 Explanation of PNSimulation's report

Before we can get into analysis of simulation's output we need to understand the format of its report. This here is an example of one such report:

```
[S] default[1][time=0][cal=[]]
  [O][1]:C1
    [N][1]:#object
      [P] counter:{(1‘10)}
      [P] result:{}
      [P] start:{(1‘10)}
      [P] temp:{}
      [T] t2:
        [precond] start(1‘x=_)
        [postcond] temp(1‘o=_)
      [T] t1:
        [precond] counter(1‘x=_)
        [precond] temp(1‘o=_)
        [postcond] result(1‘y=_)
```

Here the `[S]` represents a simulation `default` is the name of the simulation, `[time=0]` represents the internal time of the simulation, and finally `[cal=[]]` represents a calendar that is currently empty. `[O]` represents object, `[N]` represents net, `[P]` represents

place and `[T]` represents transition. In case of transition we can also see PNTransgoal `[precond] start(1‘x=_)` where `[precond]` is a type of PNTransgoal, `start` is a name of a place and `1‘x=_` where 1 represent multiplicity and `x` a variable to which a value will be assigned.

6.2 Testing models

Overview of all models used for performance testing. All of these models are implemented in class Models as methods, meaning that there are three methods model1, model2 and model3.

6.2.1 Overview of model1

For this model our root is gonna be one instance of class C1. This is very basic model has a single transition that has a input from place p1, it takes the value from place p1 and puts two of those values in place p2.

```
class C1 is_a PN
object
  place p1(2‘10)
  place p2()
  trans t1
    precond p1(1‘x)
    postcond p2(2‘x)
```

Listing 6.1: model1 class C1.

6.2.2 Overview of model2

For this model our root is one instance of class C1. The class C1 has one transition t1 that has one input (x) from place counter. The transition has also two actions associated with it, the first one creates a new instance of class C2 and assigns it to variable o. The next action calls C2's increment method with x as an argument. The increment method adds 1 to the passed argument and returns the value.

```
class C1 is_a PN
object
  place start(1‘10)
  place result()
  trans t1
    precond counter(1‘x)
    action {
      o = C2 new.
      y = o increment: x.
    }
    postcond result(1‘y)
```

Listing 6.2: model2 class C1.

```
class C2 is_a PN
object
method increment: p1
  place p1()
  place return()
  trans t2
    precond p1(1‘x)
    action {
      res = x + 1.
    }
    postcond return(1‘res)
```

Listing 6.3: model2 class C2.

6.2.3 Overview of model3

For this model our root is one instance of class C1. This will serve as a stress test for the mark phase of garbage collection as it contains a lot of references. The class C1 creates 10 new instances of class C2 and calls the increment method. Class C2 also creates 10 new instances of class C2 and calls the increment method, this works in a recursive way, meaning that more and more objects will be created with each step of the simulation.

```
class C1 is_a PN
object
 place start(10‘42)
 place end()
 trans t1
   precond start(1‘x)
   action {
     o = C2 new.
     y = o increment: x.
   }
   postcond end(1‘42)
```

Listing 6.4: model3 class C1.

```
class C2 is_a PN
object
 place start(10‘42)
 place end()
 trans t2
   precond start(1‘x)
   action {
     o = C2 new.
     y = o increment: x.
   }
   postcond end(1‘42)
method increment: in
 place in()
 place return()
 trans t22
   precond in(1‘x)
   action {
     y = x + 1.
   }
   postcond return(1‘y)
```

Listing 6.5: model3 class C2.

6.3 Testing of correctness

The purpose of this section is to validate whether our garbage collectors can accurately identify objects that are no longer accessible. I will be comparing PNSimulation's reports at crucial steps to see whether the garbage collector can detect unreachable objects and unnecessary roots correctly. All these test are using models from the previous section (the number of the test corresponds to the number of the model). I will be testing all of the garbage collectors but only showcasing the mark-sweep implementation here, since testing all three would result in a lot of repetition, without much extra information. Furthermore for all of these test I will be setting `stepsToGc` to 1 in order to trigger the GC every step.

6.3.1 Results of test1

Here on left we can see the report output for no garbage collection and on the right we see output with garbage collection. In this particular case we are testing whether we can correctly delete GC roots. You might notice that on the right side that in step 2 the object

gets already deleted, that is because right after `t1` is finished garbage collection is triggered. The report is generated only after garbage collection is done.

```
+++ step: 1
[S] default[1][time=0][cal=[]]
  [O][1]:C1
    [N][1]:#object
      [P] p1:{(1‘10)}
      [P] p2:{(2‘10)}
      [T] t1:
        [precond] p1(1‘x=_)
        [postcond] p2(2‘x=_)
---
+++ step: 2
[S] default[1][time=0][cal=[]]
  [O][1]:C1
    [N][1]:#object
      [P] p1:{}
      [P] p2:{(4‘10)}
      [T] t1:
        [precond] p1(1‘x=_)
        [postcond] p2(2‘x=_)
```

Listing 6.6: test1 with no GC

```
+++ step: 1
[S] default[1][time=0][cal=[]]
  [O][1]:C1
    [N][1]:#object
      [P] p1:{(1‘10)}
      [P] p2:{(2‘10)}
      [T] t1:
        [precond] p1(1‘x=_)
        [postcond] p2(2‘x=_)
---
+++ step: 2
[S] default[1][time=0][cal=[]]
```

Listing 6.7: test1 with GC

6.3.2 Resutls of test2

Same as in the previous test on the left no GC on the right with GC. This test has two purposes, first check if our garbage collector will remove a root with a thread in progress, which it does not, the second is whether we can detect that `[O][2]` is no longer reachable. From the result of the test we can see that both the root `[O][1]` and the object created by our root get removed from the simulation.

```
+++ step: 2
[S] default[1][time=0][cal=[]]
  [O][1]:C1
    [N][1]:#object
      [P] result:{}
      [P] start:{}
      [T] t1:
        [precond] start(1‘x=_)
        [postcond] result(1‘y=_)
        [THREAD][3][1]
        [FINISHINGOP]
        [{x=1, self={[O][1]:C1},
        o={[O][2]:C2}}][_res=2]
  [O][2]:C2
    [N][1]:#object
    [N][2]:#increment:
      [P] p1:{}
      [P] return:{(1‘2)}
      [T] t11:
        [precond] p1(1‘x=_)
        [postcond] return(1‘y=_)
---
+++ step: 3
[S] default[1][time=0][cal=[]]
  [O][1]:C1
    [N][1]:#object
      [P] result:{(1‘2)}
      [P] start:{}
      [T] t1:
        [precond] start(1‘x=_)
        [postcond] result(1‘y=_)
  [O][2]:C2
    [N][1]:#object
```

Listing 6.8: test2 with no GC

```
+++ step: 2
[S] default[1][time=0][cal=[]]
  [O][1]:C1
    [N][1]:#object
      [P] result:{}
      [P] start:{}
      [T] t1:
        [precond] start(1‘x=_)
        [postcond] result(1‘y=_)
        [THREAD][3][1]
        [FINISHINGOP]
        [{x=1, self={[O][1]:C1},
        o={[O][2]:C2}}][_res=2]
  [O][2]:C2
    [N][1]:#object
    [N][2]:#increment:
      [P] p1:{}
      [P] return:{(1‘2)}
      [T] t11:
        [precond] p1(1‘x=_)
        [postcond] return(1‘y=_)
---
+++ step: 3
[S] default[1][time=0][cal=[]]
```

Listing 6.9: test2 with GC

6.3.3 Results of testHold

For the last test I used a test that was already a part of PNtalk, here is its model 6.10.

```
class C1 is_a PN
object
  place counter(1‘10)
  place result()
  trans t1
    precond counter(1‘x)
```

```
action {
  y = self hold: 10
}
postcond p2(1‘y)
```

Listing 6.10: testHold model (taken from PNtalk's source code comments).

In this test we take a look at whether or not can our garbage collector handle native methods. The transition t1 executes a native method that sleeps for 10 units of time. We can also notice that it is the first test that modified the internal time of the simulation.

```
+++ step: 2
[S] default[1][time=10][cal=[]]
  [O][1]:C1
    [N][1]:#object
      [P] counter:{}
      [P] result:{}
      [T] t1:
        [precond] counter(1‘x=_)
        [postcond] result(1‘y=_)
        [THREAD][3][0]
        [FINISHINGOP]
        [{x=10, self={[O][1]:C1},
        _1=10}][_res=null]
    [NATIVE][2]:hold:
---
+++ step: 3
[S] default[1][time=10][cal=[]]
  [O][1]:C1
    [N][1]:#object
      [P] counter:{}
      [P] result:{(1‘null)}
      [T] t1:
        [precond] counter(1‘x=_)
        [postcond] result(1‘y=_)
```

Listing 6.11: testHold with no GC

```
+++ step: 2
[S] default[1][time=10][cal=[]]
  [O][1]:C1
    [N][1]:#object
      [P] counter:{}
      [P] result:{}
      [T] t1:
        [precond] counter(1‘x=_)
        [postcond] result(1‘y=_)
        [THREAD][3][0]
        [FINISHINGOP]
        [{x=10, self={[O][1]:C1},
        _1=10}][_res=null]
    [NATIVE][2]:hold:
---
+++ step: 3
[S] default[1][time=10][cal=[]]
```

Listing 6.12: testHold with GC

6.4 Metrics

For testing metrics I have decided to go with:

1. Time spent collecting garbage: This is because it has by far the biggest impact on user experience as the simulation has to be paused while garbage collection is running. This time is further split into the time spent marking and sweeping, this allows for greater insight into the performance of the garbage collector.

2. Size of PNSimulations object list: I have decide to go with this metric instead of something like current memory usage because java itself runs in managed runtime

environment and we have no idea when java triggers its own garbage collection, for those reasons I think including current memory usage would be more confusing than beneficial.

3. Amount of objects created: this will allow us to see how many objects have been destroyed by our GC, during the test

4. Total time: This refers to the total time of running the test, this allows us to compare how much time is spent simulating in comparison to how much time is spent on garbage collection.

6.5 Hardware

Specifying the hardware used for benchmarks is crucial as it ensures the reproducibility and comparability of performance measurements, provides context for interpreting results, offers insights into potential optimizations.

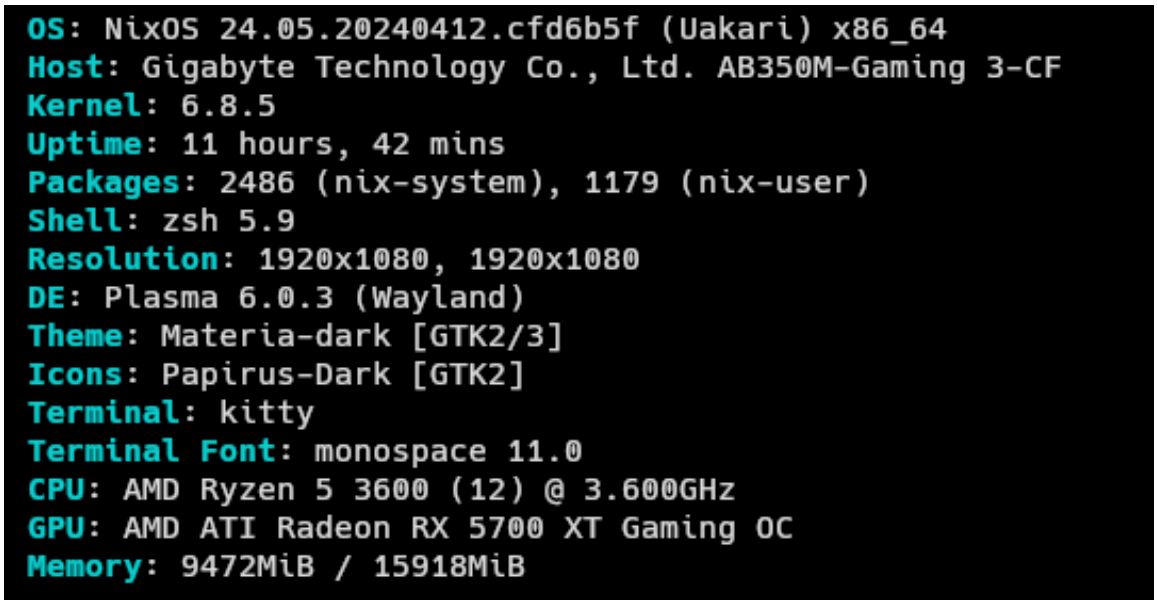

Figure 6.1: Testing hardware specifications.

6.6 Basic testing structure

All of our tests are implemented inside a class GCTest (Do not confuse these with tests from PNVM.java). The basic structure of a test is that it first creates the desired garbage collector, after which we initialize a new simulation with our create GC as an argument. Following that we loop over the model we wanna test/benchmark, the number of loops is determined by the GCTest's member variable `numOfModels`. After we are done creating our models we start a new loop, that performs the steps inside of the simulation, the maximum number of steps is give be the member variable `numOfSteps`.

```
GarbageCollectorTest gc = new GarbageCollectorTest(1);
PNSimulation sim = new PNSimulation("default", 1, gc);
```

```
for (int i = 0; i < numOfModels; i++) {
    Models.model1(sim);
}

for (int i = 0; i < numOfSteps; i++) {
    doStep(sim, i);
}
```

Listing 6.13: Basic test structure (this particular example is test1).

6.7 Performance testing

The performance testing section delves into the evaluation and analysis of the garbage collector's performance characteristics under different workloads and scenarios, providing valuable insights into its behavior and resource utilization.

There are in total 9 tests, each of the models is tested on each of the garbage collector implementations. That means there are three variations of test1:

- `test1` - this tests the basic mark-sweep implementation
- `test1P` - this tests the parallel mark-sweep implementation
- `test1G` - this tests the generational GC implementation

The number in test1 corresponds to the model that was used for the test, so for test1 it would be model1, for test2 it would be model2.

Table 6.1 contains all parameters that are fed to the garbage collector's constructor. The x in `testx-01` stands for the number of the test, so `test1-01` and `text2-01` use the same parameters for the GC constructor, also the `testx-01` GC parameters apply to the parallel and generational version of the tests.

Test	stepsToGc	maxThreads	promotion	oldGenMaxSize
noGC	-	-	-	-
testx-01	1	10	3	10
testx-02	20	20	10	100

Table 6.1: Garbage collector parameters for tests.

Before we get into the results quick explanation of the columns of table 6.2, `NoM` and `NoS` stand for `numberOfModels` and `numberOfSteps`. The total column contains the total duration of the test (this includes inserting models into the simulation), sweep column contains the amount of time spent sweeping, column mark contains the marking duration, the column Objs represents the amount of PNObjects in PNSimulation's memory after the test has finished and finally the column Objs created shows the amount of objects created during the run of the simulation.

6.7.1 Results of testx-01

As we can see in table 6.2 the test1-01 (mark-sweep implementation) is the overall best performer in this test. The parallel implementation struggles with tasks that have a low amount of objects in simulation's memory, but things improve once we increase the number of models, namely they improve for the mark phase duration as it is 4 seconds lower than mark-sweep implementation. The generational implementation does the best in the test with low amount of models/steps, this is because this particular model cannot trigger its full GC. With higher amount of models/steps the generational implementation struggles quite hard as it does not remove any roots, meaning the list of objects never gets smaller.

Test	NoM	NoS	Total[ms]	Sweep[ms]	Mark[ms]	Objs	Objs created
noGC1	100	100	66	-	-	100	0
test1-01	100	100	49	0.82	4.06	51	0
test1P-01	100	100	165	81.48	69.62	51	0
test1G-01	100	100	30	0.04	3.96	100	0
noGC1	10000	10000	19189	-	-	10000	0
test1-01	10000	10000	12717	768.12	11175.82	5001	0
test1P-01	10000	10000	14817	6523.01	7770.19	5001	0
test1G-01	10000	10000	55098	3.51	30309.99	10000	0

Table 6.2: Results of test1.

Table 6.3 shows similar results to the first table, at least when comparing mark-sweep and parallel implementations. The generational implementation shows great improvement in both low number of models/steps and higher number of models/steps. When it comes to parallel marking it is more than twice as fast as the mark-sweep implementation.

Test	NoM	NoS	Total[ms]	Sweep[ms]	Mark[ms]	Objs	Objs created
noGC2	100	100	111	-	-	199	99
test2-01	100	100	126	1.77	9.04	199	99
test2P-01	100	100	135	38.95	43.13	199	99
test2G-01	100	100	38	1.62	5.63	199	99
noGC2	10000	10000	64071	-	-	19999	9999
test2-01	10000	10000	142962	4271.76	54834.88	19999	9999
test2P-01	10000	10000	110671	5105.35	23462.99	19999	9999
test2G-01	10000	10000	172715	16223.24	62624.76	19999	9999

Table 6.3: Results of test2.

Once again the table 6.4 shows the same pattern. With generational implementation struggling hard when it comes to sweep times with higher amount of models/steps.

Test	NoM	NoS	Total[ms]	Sweep[ms]	Mark[ms]	Objs	Objs created
noGC3	100	100	73	-	-	199	99
test3-01	100	100	91	1.71	12.97	199	99
test3P-01	100	100	99	38.36	39.00	199	99
test3G-01	100	100	19	1.54	5.00	199	99
noGC3	10000	10000	14665	-	-	19999	9999
test3-01	10000	10000	92437	4176.68	67900.72	19999	9999
test3P-01	10000	10000	74595	5724.37	43910.64	19999	9999
test3G-01	10000	10000	109477	16440.57	70182.24	19999	9999

Table 6.4: Results of test3.

6.7.2 Results of testx-02

The most curious results 6.5 from this testing is that the parallel implementation in couple of cases despite having much higher sweep and mark time the total time is lower. I tried looking into why this is but unfortunately due to a lack of time I was not able to figure it out. For the rest of the results, we can see that the advantage of parallel's marking at higher amount of models/step has weakened this is most likely due to the reduction in amount of models. The generational implementation is now overall more in line with the other implementations and in test3G-02 it has done exceptionally well, this indicates that the generational implementation has its use cases and with tweaking some of its parameters we could possibly even get better results.

Test	NoM	NoS	Total[ms]	Sweep[ms]	Mark[ms]	Objs	Objs created
noGC1	100	1000	354	-	-	100	0
test1-02	100	1000	63	0.11	1.06	0	0
test1P-02	100	1000	49	12.90	18.04	0	0
test1G-02	100	1000	247	0.06	2.03	100	0
noGC1	5000	10000	19014	-	-	5000	0
test1-02	5000	10000	591	17.14	185.82	10	0
test1P-02	5000	10000	1392	575.90	619.92	10	0
test1G-02	5000	10000	21597	1.07	612.58	5000	0
noGC2	100	1000	430	-	-	200	100
test2-02	100	1000	134	0.26	2.41	0	100
test2P-02	100	1000	100	15.24	19.67	0	100
test2G-02	100	1000	221	0.22	3.52	200	100
noGC2	5000	10000	47059	-	-	10000	5000
test2-02	5000	10000	25999	57.43	1003.82	42	5000
test2P-02	5000	10000	28050	636.23	846.95	42	5000
test2G-02	5000	10000	28378	217.34	1011.76	162	5000
noGC3	100	1000	373	-	-	1099	999
test3-02	100	1000	407	3.39	23.49	1080	999
test3P-02	100	1000	325	58.81	70.22	1080	999
test3G-02	100	1000	174	6.95	14.84	1080	999
noGC3	5000	10000	13213	-	-	14999	9999
test3-02	5000	10000	17026	123.89	2544.44	14980	9999
test3P-02	5000	10000	15962	602.17	1237.04	14980	9999
test3G-02	5000	10000	17186	688.11	2458.82	14980	9999

Table 6.5: All results.

Chapter 7

Conclusion

In conclusion we went over the different approaches to memory management. Summarized many different kinds of garbage collection algorithms, looked at their respective strength and weaknesses. Analyzed PNtalk's structure and implementation, identified all the problems we had to solve in order to implement garbage collection. Used all the knowledge from previous chapter to implement three different garbage collection algorithms, detailed changes that were made to PNtalk. After this we tested and benchmarked all three implementations, we tested for correctness and performance. I think the the performance testing has shown that garbage collection contributes greatly to performance by identifying and freeing up unused memory periodically.

Overall I would say that my work has contributed to development of PNtalk in positive ways, mainly in cases where there are large and complex Object-Oriented Petri Nets at play. However, challenges remain, and avenues for future research include further optimization, fine-tuning, and exploration of alternative garbage collection strategies to meet evolving demands and requirements in PNtalk.

Bibliography

[1] Heller, M. *What is garbage collection? Automated memory management for your programs* online. 2023. Available at: https://www.infoworld.com/article/3685493/what-is-garbage-collection-automated-memory-management-for-your-programs.html. [cit. 2024-04-24].

[2] Janoušek, I. V. *Modelování objektů Petriho sítěmi.* Brno, CZ, 1998. Disertační práce. VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ Fakulta elektrotechniky a informatiky. Available at: https://www.fit.vutbr.cz/~janousek/publications/phdthesis.pdf.

[3] Kočí, R.; Janoušek, V. and Zbořil, F. Object Oriented Petri Nets - Modelling Techniques Case Study. *International Journal of Simulation Systems, Science & Technology*, 2010, vol. 10, no. 3, p. 32–44. ISSN 1473-8031. Available at: https://www.fit.vut.cz/research/publication/9194.

[4] Richard Jones, A. H. and Moss, E. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* 1st ed. Chapman and Hall/CRC, 2011. ISBN 978-1-4200-8279-1.

Appendix A

Contents of the included memory medium

- src - directory that contains the PNtalk source code with my contributions mentioned in a readme file.
- thesis - directory that contains source files to the text of my thesis.
- thesis.pdf - this thesis in pdf format.
- poster.pdf - poster for this thesis.