國立東華大學資訊工程學系
碩士論文

在行動者模型上的垃圾回收演算法及其在嵌入式系統上之應用

A New Garbage Collection Algorithm for the Actor Model and Its Application to Embedded Systems

研究生：涂文凱

指導教授：雍　忠 博士

中華民國九十五年七月

A New Garbage Collection Algorithm for the Actor Model and Its Application to Embedded Systems

by

Wen-Kai Tu

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science and Information Engineering

National Dong Hwa University

July 2006

Approved: _____

Chung Yung

國立東華大學
學位論文授權書

※說明※
本授權書請撰寫並簽名後，裝訂於紙本論文書名頁之次頁。

本授權書所授權之論文為本人在國立東華大學＿＿資 訊 工 程＿＿系所＿＿＿＿＿＿組

＿九十四＿學年度第＿二＿學期取得＿碩＿士學位之論文。

論文名稱： 在行動者模型上的垃圾回收演算法及其在嵌入式系統上之應用
(A New Garbage Collection Algorithm for the Actor Model and Its Application to Embedded Systems)

指導教授姓名：雍忠 教授

學生姓名：涂文凱

學號：69321037

本人具有著作財產權之上列論文全文資料，基於資源共享理念、回饋社會與學術研究之目的，非專屬、無償授權國立東華大學及國家圖書館，得不限地域、時間與次數，以微縮、光碟或數位化等各種方式重製散布、發行或上載網路，提供讀者非營利性質之線上檢索、閱覽、下載或列印。

上述數位化公開方式限：

■校內、校外公開。

□校內公開，校外因

1.上列論文申請專利（案號： ），請於 3 年後公開。

2.上列論文內容隱私權之需要（請指導教授附函說明特殊原因），請於 3 年後公開。

□校內公開，校外因＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿，請於 1 年後公開。

授權內容均無須訂立讓與及授權契約書，授權之發行權為非專屬性發行權利。依本授權所為之收錄、重製、發行及學術研發利用均為無償。數位化公開方式若未勾選，本人同意視同授權校內、校外公開。

簽　　名 (親筆正楷)		日期	中華民國　　年　　月　　日

0950215 表 LB702

Acknowledgements

First of all, I must offer my heartfelt thanks to my advisor, Professor Chung Yung. In the process of writing my thesis, Professor Yung supervises my progress, reads and revises my drafts carefully and gives me insightful remarks. He also teaches me how to study algorithm research and to bring up my warm heatedness.

In addition, I would like to thank my good friends Yi-Fen Lin, Sha Luo, Gang-Lin Wu, Syue-Ping Yang, Yi-Ming Chang and laboratory members, Ming-Sian Lin, Ching-Ho Cheng, Ying-Chen Li, Bing-Liang Shih and Fowei Yun. They give me the hole support to my thesis. We go through happiness and sorrow. I am so glad to be friend with them. Finally, I want to give this honor to my family. They are my mainstay.

A New Garbage Collection Algorithm for the Actor Model and Its Application to Embedded Systems

Wen-Kai Tu

Advisor: Chung Yung

Abstract

This thesis proposes a new garbage collection algorithm for the actor model, and its application to embedded systems is included. In the actor model, the universe consists of autonomous computational agents called *actors* that encapsulate data as well as some primitive processing power to manipulate data. Garbage collection is an important service which provides error-free and automatic management of system resources. In recent years, the application software for embedded systems is more and more complicated. According to the description by Bacon et al., the requirements on garbage collectors in the embedded environments are: code size, memory overhead, compaction, reliability, smooth performance and speed. The focus of this thesis is to develop a new garbage collection algorithm for the actor model that can be applied to embedded systems without difficulty.

Based on the garbage of actors defined by Nelson, we propose the Levelled-Visit algorithm to collect garbage. We define the concept of visit-level, in which all actors have the same color. Improved from the Push-Pull algorithm and the Is-Black algorithm for garbage collection proposed by Kafura et al.,

our algorithm eliminate the repeated visits to actors.

This thesis also includes the proof of correctness and the complexity analysis for the proposed algorithm. The time complexity of our algorithm is O(N^2), which the same as the Push-Pull and Is-Black algorithms. We have implemented our algorithm in the Java^{TM} Platform, Micro Edition(J2ME). J2ME is the most ubiquitous application platform for mobile embedded devices. We experiment on the time required by various garbage collection algorithms. The execution time of garbage collection is reduced from 2.128% to 64.98% by our algorithm over the Push-Pull(33.191% $\sim$ 64.98%) and Is-Black(2.128% $\sim$ 25.129%) algorithms.

在行動者模型上的垃圾回收演算法及其在嵌入式系統上之應用

涂文凱

指導教授 : 雍 忠 教授

摘要

本篇論文我們在行動者模型上面提出一個新的垃圾回收演算法，並將之應用在嵌入式系統上。在行動者模型(actor model)中，我們稱行動者(actor)是一個可以自動計算的代理人(agent)，而這些代理人的集合就稱為行動者模型。這些行動者封裝控制執行緒以及狀態在一個實體裡面，以達到可以並行處理的機制。垃圾收集(garbage collection)是一個非常重要的系統服務，它提供系統資源自動管理的服務。而在最近這幾年，在嵌入式系統上的軟體應用越來越複雜。根據Bacon等人在嵌入式的垃圾回收機制裡面所提出的需求，包括程式碼大小、記憶體負載、壓縮、可靠性、效能動態調整以及速度，都是近幾年來研究的方向。本篇論文著重在以行動者模型上發展一個新的垃圾回收演算法，並且將之部署在嵌入式系統上。我們根據Nelson所定義的垃圾行動者(garbage of actor)，提出一個稱為階層拜訪(Levelled Visit)的演算法去收集這些垃圾。我們定義了拜訪階層

(visit level)的概念，那是一種在此階層下的所有行動者，都擁有相同顏色的屬性。我們也試圖改善Kafura所提出的Push-Pull以及Is-Black演算法，消除這兩個演算法中會出現重複拜訪的actors的現象。本篇論文也包含我們演算法的正確性證明以及複雜度分析。我們演算法的時間複雜度是$O(N^2)$，和Push-Pull演算法以及Is-Blcak演算法具有相同的時間複雜度。本論文將我們所提出的演算法，使用Java™ Platform, Micro Edition(J2ME)來實作。J2ME是發展行動式嵌入式系統的一個應用平台。 最後在我們的實驗當中，和Push-Pull演算法比較，我們減少 33.191%到 64.98%的時間消耗；和Is-Black演算法比較，我們減少 2.128%到 25.129%的時間消耗。

Contents

List of Figures

Chapter 1

Introduction

This thesis proposes a new garbage collection algorithm for the actor model. In the actor model, the universe consists of autonomous computational agents called *actors* that encapsulate data as well as some primitive processing power to manipulate data. Nelson and Kafura et al. present garbage collection algorithm of actors[NE89, KWN90]. Garbage collection is an important service which provides error-free and automatic management of system resources. In recent years, the application softwares for embedded systems which is more and more complicated. According to the description by Bacon et al. the requirements on garbage collectors in the embedded environments are: code size, memory overhead, compaction, reliability, smooth performance and speed[BA04]. The focus of this thesis is to develop a new garbage collection algorithm for the actor model that can be applied to embedded systems without difficulty.

Actors extend sequential objects by encapsulating a thread of control along

with procedures and data in the same entity; thus actors provide a unit of abstraction and distributed in concurrency. Actors are inherently concurrent and autonomous enabling efficiency in parallel execution[WG95] and facilitating mobility[GN99]. The actor model and languages provide a useful framework for understanding and developing open distributed systems. For example, actor systems have been used for fault-tolerance and distributed artificial intelligence.

Garbage collection in actors has some similarities with garbage collection in *passive*[1] object-oriented systems but has some important differences too. The problem of garbage collection of actors is slightly different from garbage collection for "passive" objects. It needs that special algorithms to be devised for garbage collecting actors.

- In passive object-oriented systems, the model of computation is that of independent threads of control manipulating the objects they can refer to. Implicit in this idea is that the threads of control by themselves are always important. Thus, an important subset of the root set consists of the objects referenced from the stacks of running threads.

- In actor systems, there is no notion of an independent thread of control manipulating data. Instead, the thread of control is encapsulated in the object itself. Conceptually, there are no stacks, registers or a separate heap (an actual implementation however may still have these). The idea

[1]We emphasize that objects in traditional oriented-object systems are "passive" as opposed to actors being "active".

that all threads of control are important to the underlying computation is no longer valid; no actor would otherwise ever become garbage because each actor is referenced by the thread of control it encapsulates.

To identify garbage in an actor system, we have to take a look at how an actor system performs some computation. Conceptually, a single actor called the root actor can be deemed to start the computation (if more than one actor exists at startup, one can designate a hypothetical "root" actor that directly references the startup actor). This actor may interact with the user and hence may be relevant to the computation at all times. As the computation proceeds, the root actor can create more actors that in turn may create other actors. Message are passed from actor to actor, in response to which some progress is made in the underlying computation. Any actor that is unable to receive a message from or send a message to the root, directly or through intermediate actors, is therefore irrelevant to the underlying computation and should be regarded as garbage. This illustrates an important difference in garbage collection between actor systems and passive object-oriented systems.

Nelson[NE89] formulated certain coloring rules to identify actor garbage. We discuss it in Chapter 2. Based on these rules, Kafura et al. present two algorithms[KWN90], Push-Pull and Is_Black algorithms. In the Push-Pull marking algorithm, there are two coroutines, a Pusher and a Puller, to move actors between black, gray and white sets. The Pusher operates on the white set and tries to push a white actor into the gray or black set depending on whether it has an acquaintance which is black or gray. The Puller pulls

acquaintances of black actors out of the gray or white set into the black set. In the algorithm Is_Black, one rule repeatedly colors black the acquaintances of black actors. A second does a depth first search from all active actors for a black actor. If a black actor is found, then the originating actor is colored black. Both these algorithms have a space complexity of O(N) and time complexity of O(N^2) where N is the number of actors in the system.

According to the description by Bacon et al., the requirements on garbage collectors in the embedded environments are:

- **Code Size**. It is imperative that the virtual machine consume as little code space as possible. As a result, many typical methods for improving collector performance are inappropriate because of the resulting complexity and concommitant increase in code size.

- **Memory Overhead**. The overhead due to collector meta-data and memory fragmentation should be kept to an absolute minimum. As a result, semi-space copying collectors [MJ60, FY69] are not an option.

- **Compaction**. Since many embedded applications run continuously for extended periods of time., the collector must be able to perform memory compaction, both to avoid arbitrary space consumption due to fragmentation, and to enable the virtual machine to release memory resources to the operating system as needed.

- **Reliability**. System failure is not acceptable. This places a premium on both simplicity and on strong enforcement of invariants within the

collector.

- **Smooth Performance**. The likelihood that the application will run in a very constricted memory space is much higher than in PC- or server-based virtual machines. Therefore, the collectors performance should degrade gracefully as memory is reduced.

- **Speed**. Within the limits of the preceding requirements, the collector should be as fast as possible. Trading a small amount of space for a large improvement in time is acceptable for some, but not all, applications.

Based on the garbage of actors defined by Nelson, we propose the Levelled-Visit algorithm to collect garbage. We define the concept of visit-level, in which all actors have the same color. Improved from the Push-Pull algorithm and the Is-Black algorithm for garbage collection proposed by Kafura et al., our algorithm eliminate the repeated visits to actors.

1.1 A Motivation Example

In this section, we provide an example to show our observation for Is_Black algorithm of garbage collection.

Figure 1.1 is an Example of the Actor Reference Graph. Actor F is the root actor, another actors A,C and E are active actor, and still another actors B and D are block actor. We will introduce the definition of kind of actor in the chapter 2. Now we just know how to determine actors A,B,C,D and E whether it's garbage actors or not.

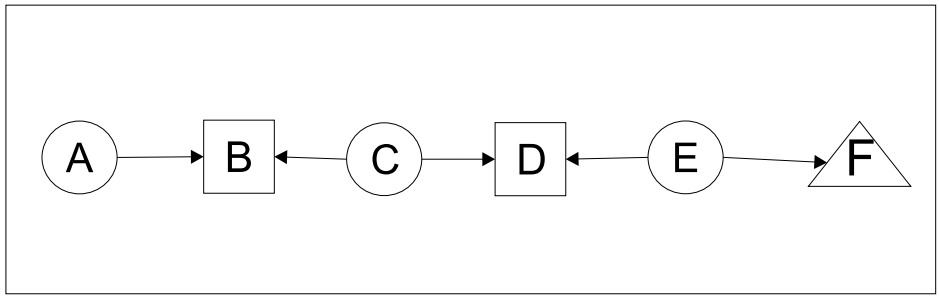

Figure 1.1: An Example of the Actor Reference Graph.

We apply the Is_Black algorithm to collect an example of Figure 1.1. In this case, first iteration of the Is_Black algorithm, the actor A,B,C,D and E will be visited, then determine the actor E and D are non-garbage. Second iteration, the actor A,B and C will be repeated visited again, then determine the actor C and B are non-garbage. Third iteration, the actor A will be repeated visited again, then determine the actor A is non-garbage finally. Note that, in this example, the actor A is repeatedly visited three times, the actors B and C are repeatedly visited two times. According to our observation, these repeated visits are redundant and can be eliminated.

The goal of this thesis is to develop a new garbage collection algorithm which eliminates the redundant repeated visits. We present the newly developed algorithm in Chapter 3.

1.2 Road-Map to the Thesis

The remainder of the thesis is organized as follows:

- Chapter 2 describes the background and related work. We explain some

important concepts in the actor model and garbage collection. The actor model of computation is presented and the distinction between garbage collection in object oriented systems and actor systems is described.

- Chapter 3 proposes the Levelled_Visit algorithm for the actor model. The algorithm is designed in a way that no repeated visit to the actors. We also give an example in this chapter.

- Chapter 4 analyzes the correctness and computational complexity of our algorithm. We first give five lemmas, and the prove the theorem in this chapter.

- Chapter 5 includes the experiments on our algorithm in the Java^{TM} Platform, Micro Edition(J2ME). We also implement Push-Pull and Is_Black algorithms in the J2ME for comparison. And, we give the experimental benchmarks.

- Chapter 6 gives the conclusions and a brief summary of the thesis.

Chapter 2

Background and Related Work

This chapter provides background for concepts in the actor model and garbage collection. The model of computation in actor systems is explained and garbage collection in both non-actor and actor systems is discussed.

2.1 Actor Systems

In the Actor model[GA86], the universe consists of autonomous computational agents called *actors* that encapsulate data as well as some primitive processing power to manipulate data. Each actor has a unique mail address which can be used to communicate with that actor. Communication between actors is asynchronous with unbounded delay but guaranteed delivery. Message may be received in an order different than sent. Message for an actor are buffer in a *mail queue.*

Computation in an actor system is message driven. Each actor process message in its mail queue one by one in the order they were queued. Processing

a message involves executing a script which is called the *behavior* of the actor.

In response to a message, an actor may *create* new actors, *send* message to actors, and *change* its state with it responds to the next message (2.1). Theses actions are implemented by extending a sequential language with the following operators:

- **create** takes a behavior description and creates an actor. The operator may take additional initialization arguments.
- **send** takes the receiver's mail address and puts a message int its mail queue.
- **ready** changes its behavior to the one specified and prepares to process the next message in the mail queue.

The actor model provides a powerful abstraction for concurrent and distributed systems. By encapsulating a thread of control along with data, a number of synchronization problems that often plague concurrent object-oriented systems are avoided.

The graph constructed from actors as nodes and their references as edges is called the *actor-reference graph*. Mail addresses may be communicated through messages leading to dynamic changes in the actor-reference graph.

2.2 Garbage collection

Garbage collection refers to the automatic reclamation of resources which have become unusable by the user program. It is employed most commonly to re-

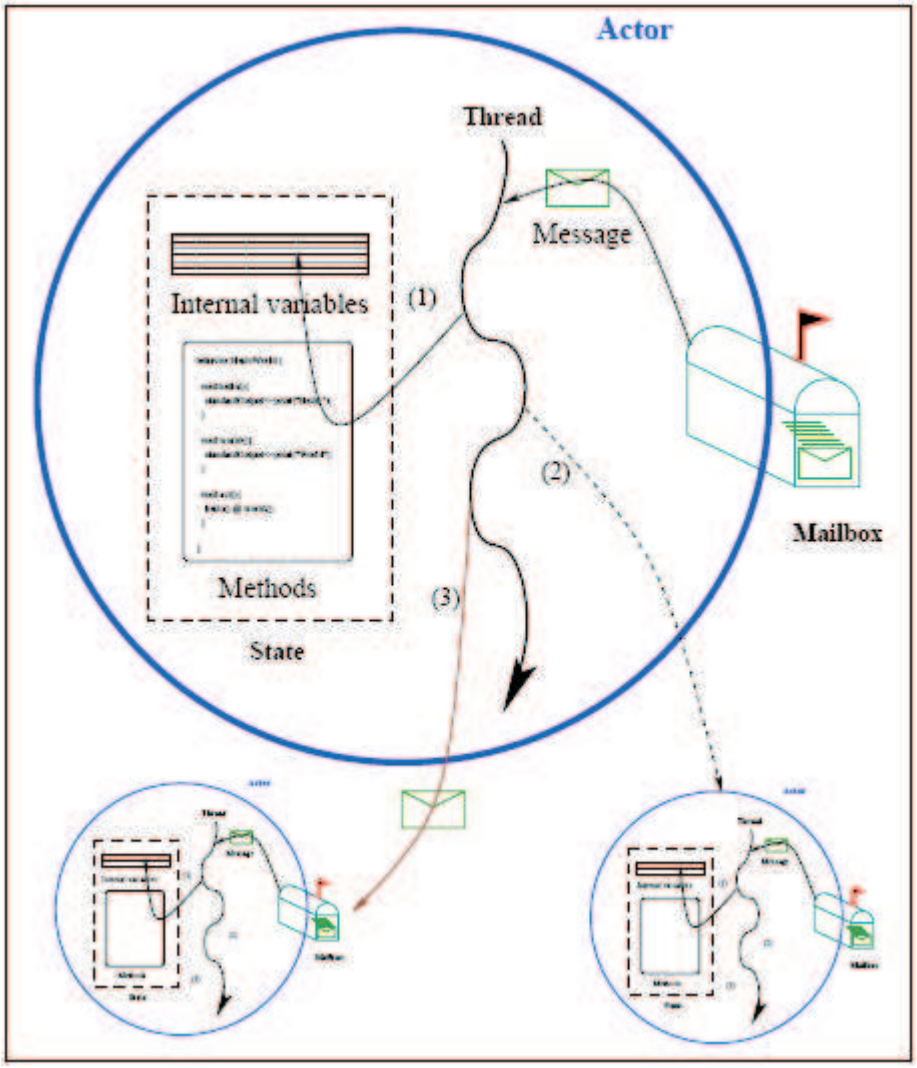

Figure 2.1: Primitive operations in the Actor Model. In response to message, an actor can:(1)change its local state, or (2) create new actors, or (3) send messages to acquaintances.

claim dynamically allocated memory. Garbage collection has been used extensively in functional languages such as Lisp and Scheme. Object-oriented languages such as Java and Smalltalk also support garbage collection.

Although garbage collection has been applied to a variety of languages, we shall focus our attention on garbage collection for object-oriented systems and actors only. The issues involved in other types of languages are analogous to those in object-oriented systems and actors.

2.3 Garbage collection in object-oriented systems

Garbage collection of the heap in an object-oriented system can be viewed as an abstract graph problem. A directed graph is formed with objects as nodes and object references as edges. We call this graph an *object-reference graph*. A subset of objects, such as statically declared objects, input-output objects, objects on the program stack and object referenced from the CPU registers, are considered to be non-garbage and are called *root* object. The problem of garbage collection is reclaiming all the objects which cannot affect the user computation. This is equivalent to finding objects which are not connected to any root object by a path through the edges of the graph. The top part of Figure 2 shows an Object-reference graph. It can be seen that objects 6, 7 and 8 are garbage.

2.4 Garbage collection in Actors

We saw a brief description of the actor model in beginning of this chapter. We now present a formal definition of garbage in actor systems. A principle features and terminology of the actor model which relate to the garbage collection problem are these:

- actor: a concurrently active object. There are no pasive entities. Each actor is uniquely identified by the address of its single mail queue.
- acquaintance: actor B is an acquaintance of actor A if B's mail queue is known to actor A.
- inverse acquaintance: if actor A is an acquaintance of actor B, then actor B is an inverse acquaintance of A.
- acquaintance list: a set of mail queue addresses including any mail queue address contained in a message on the actors mail queue address contained in a message on the actors mail queue or int transit to the mail queue. This accounts for delays in message processing.
- topology: actors may be dynamically created and actors may be dynamically bound (i.e., acquaintances can be passed at run-time through mail messages).
- behavior: a thread of execution which an actor. There may be many active threads within an actors, each thread processing a different mail message.

- blocked actor: an actor all of whose behaviors are blocked.
- active actor: an actor with at least one active behavior.
- root actor: an actor designated as being "always useful." Examples of root actors are those which have the ability to directly affect real-world through sensors, actuators, I/O devices, users, etc.

The *root set* of actors is the set of actors which directly communicate with the external environment and are always considered non-garbage. An actor which is either processing a message or has messages pending in its mail-queue is called an *active* actor. An actor which is not unblocked is call *blocked* actor.

A blocked actor which is not connected by the recursive closure of the inverse acquaintance relation to an unblock actor is called a *permanently blocked* actor. The set of live actors is then define recursively as:

1. A root actor is live.
2. Every forward acquaintance of a live actor is live.
3. Every inverse acquaintance of a live actor which is not permanently blocked is live.

An actor which is not live is garbage. A garbage actor is one which is[KWN90]:

1. not a root actor, and
2. cannot potentially receive a message a message from a root actor, and

3. cannot potentially send a message to a root actor.

In this definition, the term "potentially" requires further clarification. An actor can send a message to a root actor only if the actor is active and has the root actor as a direct acquaintance. There is a set of transformations that can change an actor reference graph from a representation of what can currently happen to what can potentially happen. The two transformations concern change in the state of an individual actor and change in the topology of the system of actors. First sending a message form an active or root actor to a blocked acquaintance allows the blocked actor to become active. This transformation reflects the ability of an actor to alter the state of another actor. The second transformation occurs when a root or active actor sends its own mail queue address or the mail queue address of one of its acquaintances to another of its acquaintances. This transformation reflects the ability of an actor to send a mail queue address, thereby changing the topology of the actor system.

Suppose an actor system has the above transformations repeatedly applied until no more transformations can be applied. Then, all actors which are not direct acquaintances of root actors in the resulting topology are garbage.

One key property of garbage actors is that they cannot become *non-garbage*. This is because actors are only determined to be garbage when there is no possibility of communication between it and a root actor. Therefore, once an actor is marked as garbage, there is no possible sequence of transformations which cause the garbage actor to become non-garbage.

Is is important to make the following two observations. First, the garbage collection problem of objects and its solution as presented in this thesis are not limit to the actor model. Similar techniques should be applicable to any object-based concurrent model which has the following general properties: encapsulated objects interact exclusively via message; the communication structure is not static; the state and acquaintances of each object can be determined. Second, the extended message passing primitives used in the actor-based language which others paper is defining, ACT++[DK90] and Actor Foundry[YA00], do not interfere the properties just noted.

Figure 2.2 shows two graphs. The top one is a reference graph graph for passive objects while the bottom one is a reference graph for a system of actors. In both graphs, an object or an actor reachable from the root is considered live(non-garbage). According to the definition of garbage actor, it can be seen that actors 7, 8, 9 and 10 are garbage in the foot part of Figure 2.2.

We see that there are some differences between the garbage collection of passive objects and actors. These differences have led to specialized algorithms being developed for actors. In the Chapter 3, we will review related garbage collection work done in uniprocessor systems and actors. We also describe garbage collection of actors is in the problem that embedded system encounters.

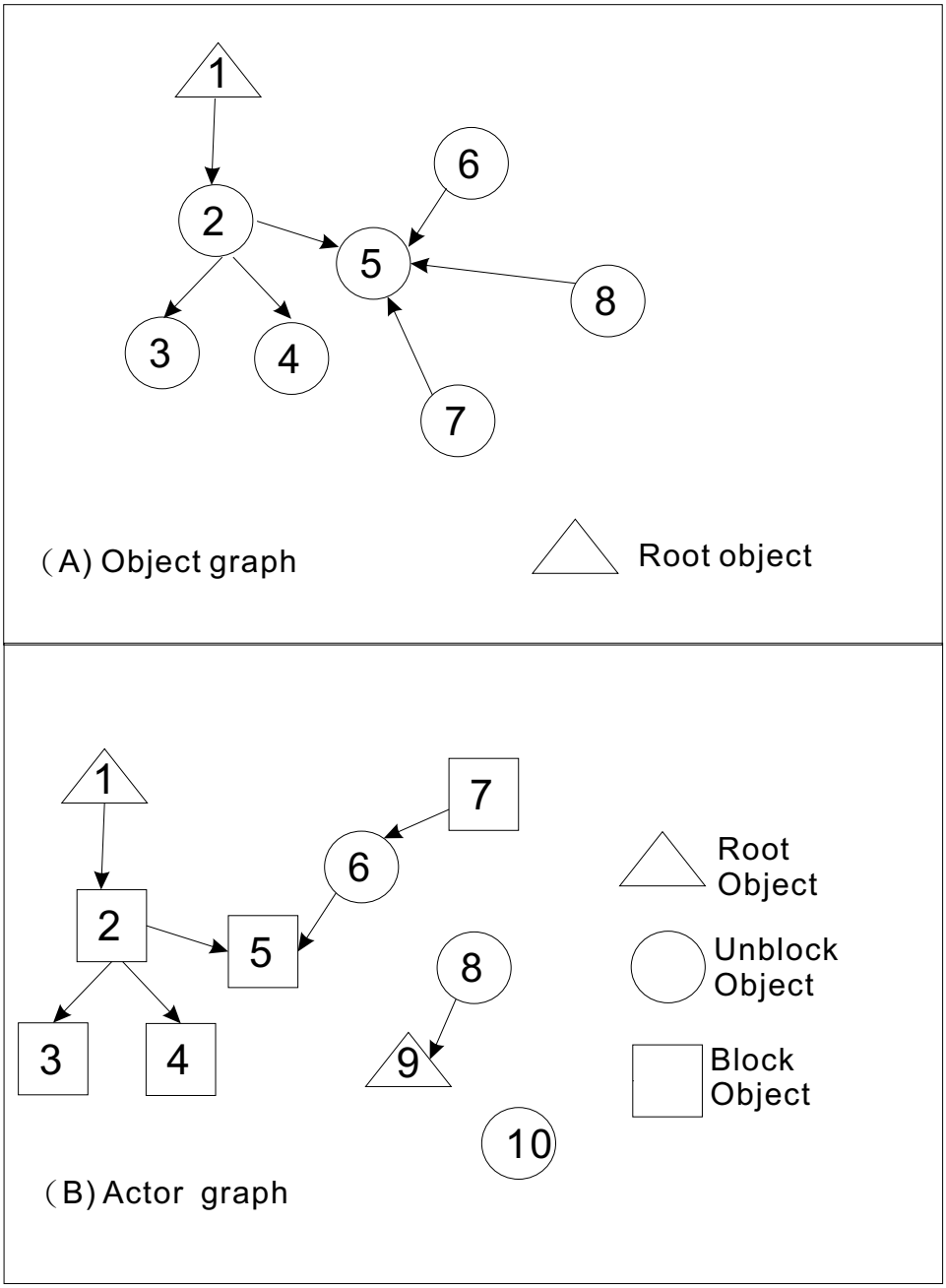

Figure 2.2: Garbage collection in passive objects and actors.

2.5 Kafura's Methodology for Garbage collection of Actors

A formal definition of garbage actors was first given by Kafura[KWN90]. The basis of the algorithms presented in his work is marking actors with three different colors that have the following meanings:

- White: It has not been shown this actor can communicate with a root actor.
- Gray: This actor is blocked but can communicate with a root actor if it can become active which has not yet been shown.
- Black: This actor is non-garbage.

Nelson[NE89] formulated certain coloring rules to identify actor garbage. There are given in figure 2.3.

Based on these rules, Kafura et al. present two algorithms, Push-Pull and Is_Black algorithms.

The Push-Pull algorithm is shown in Figure 2.4. The Is_Black algorithm is shown in Figure 2.5.

Figure 2.6 shows an actor configuration which demonstrates some features of the "Is_Black" algorithm. At the start of the algorithm, actor F is colored black because it is a root actor, and all other actors are colored white. The algorithm begins by coloring actor G black because it is an acquaintance of black actor.

Algorithm: Nelson's Coloring Rules

1. All actors are colored white, with the exception of root actors which are colored black.
2. Repeat the following rules until no more markings are made:
   (a) Color black all acquaintances of black actors.
   (b) Color black all inverse acquaintances of black actors if the inverse acquaintance is not blocked.
   (c) Color black all inverse acquaintances of black actors if the inverse acquaintance is blocked.
   (d) Color black all inverse acquaintances of gray actors if the inverse acquaintances is not blocked.
   (e) Color gray all inverse acquaintances of gray actors if the inverse acquaintances is blocked.
3. Actors that are black are not garbage; all other are garbage.

Figure 2.3: Algorithm : Nelson's Coloring Rules

The algorithm next does a depth first search from active actor A. Note the cycle between actors A and B. The algorithm applies the function Is_Black to actor B, which in turn, queries actor A. The query from actor A returns false, because it has already been visited. When actor B queries actor C, it returns false because it is not black, nor does it have any black acquaintances. Therefore, the depth first search from actor A failed, so it remains white.

The algorithm next does a depth first search from active actor D. It eventually encounters actor F, which is colored black, so actors D and E are colored black. Actor E was colored black because it was on the search path. Because an actor was colored, the first step of the algorithm is repeated. This causes actor C to be colored black because it is an acquaintance of black actor D.

Algorithm: Push-Pull

BEGIN **Initialization**
  All root actors are placed in the black set.
  All other actors are placed in the white set.
  Resume Puller
END **Initialization**

BEGIN **Puller**
  FOR[each actor in the black set not yet examined]
    placed non-black acquaintances of the actor in the black set
  END FOR
  resume Pusher
END **Pusher**

BEGIN **Pusher**
  FOR[each actor in the white set]
    CASE: actor is active and an acquaintance is black or gray
      $->$ place actor in black set
    CASE: actor is blocked and an acquaintance is black or gray
      $->$ place actor in gray set
  END FOR
  IF[any actors were placed in the black or gray set]
    THEN resume Puller
    ELSE Termination
END **Pusher**

Termination:
  All actors which are not black are garbage

Figure 2.4: Algorithm: Push-Pull

Algorithm: Is_Black

All anchors are colored black, all other actors are colored while pass=0
pass = 0
BEGIN **Is_Black**
  REPEAT
    increment pass
    color black all acquaintances of black actors
    FOR all white active actors DO Depth_First(actor,pass)
  UNTIL[no new marking are made]
END **Is_Black**

BEGIN **Depth_First(actor, pass)**
  IF [actor == black] RETURN true
  ELSE IF [actor.visit == pass] RETURN false
  ELSE
    actor.visit = pass
    FOR [each acquaintance of actor]
      IF[depth_first(acquaintance, pass)==true]
        color actor black
        RETURN true
      END IF
    END FOR
  END IF
  RETURN false
END **DEPTH_Fisrt**

Termination:
  All non-block actors are garbage

Figure 2.5: Algorithm: Is_Black Algorithm

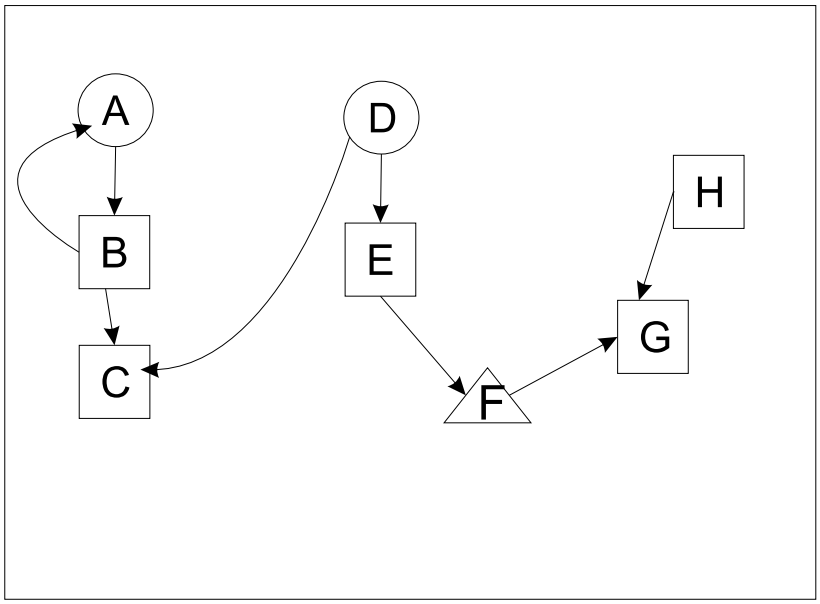

Figure 2.6: Using Is-Black Algorithm to Collect Garbages of Actor Systems.

Next, a depth first search is done from active actor A. This time the depth first search finds black actor C, so actor A is colored black.

Since an actor was again darkened, the algorithm is repeated. The first step colors actor B black, because it is an acquaintance of black actor A. The algorithm repeats itself once more, but no more colorings are done. At the termination, all actors except for actor H, which is the only garbage actor, are colored black.

Above is an introduction of the background and related work. In next chapter, we discuss the concept and details of our algorithm.

Chapter 3

A New Garbage Collection Algorithm for the Actor Model

In this chapter, we discuss our algorithm. We propose a new algorithm for collect garbage actors, the algorithm is a visit-level based garbage collection algorithm. In our algorithm, we define the visit-level is: if the actor $u \in V$(the actors of actor reference graph), the actor $v \in V$ and $(u,v) \in E$(the edges of actor reference graph), then (a) u is an active actor, and (b) v is not visited. We say that u and v are in the same visit-level. The actors in the same visit-level have the same color.

In this thesis, we define each actors of visit-level 1 is black. Our goal is that collect "actors of the color different from color of visit-level 1". If color of another visit-level has the same as color of visit-level 1, then each actors in this another visit-level are black. Otherwise, if color of another visit-level has not the same as visit-level 1, then each actors in this another visit-level is

white. In our algorithm, all the white actors computed by the our algorithm are garbage. We call the algorithm is Levelled_Visit algorithm.

Road-map of this chapter. Section 3.1 gives the definitions and notations needed for the algorithm. Section 3.2 discuss to detail of Levelled_Visit algorithm. Finally, we give an example.

3.1 Definitions and Notations

In this section, we define an element and attribute of Actor system. And, we define some data structure, that will be used in our algorithm.

1. **Actor**: an element in the Actor System.
2. **Root Actor**: The actor which directly communicates with the external environment and is always considered non-garbage.
3. **Active Actor**: An actor which is either processing a message or has messages pending in its mail-queue is called an active actor.
4. **Blocked Actor**: An actor which is not active and not a root are calling a blocked actor.
5. **A garbage actor**, we use the definition of kafura et al.[KWN90], is one which is:
   (a) is not a root actor, and
   (b) cannot potentially receive a message a message from a root actor, and

(c) cannot potentially send a message to a root actor.

6. **A non-garbage actor** which

   (a) is a root actor, or

   (b) can potentially receive a message a message from a root actor, or

   (c) can potentially send a message to a root actor.

In this definition, the term "potentially" requires further clarification[KWN90]. An actor can send a message to a root actor only if the actor is active and has the root actor as a direct acquaintance.

There is a set of transformations that can change an actor graph from a representation of what can currently happen to what can potentially happen.

The two transformations concern change in the state (ready, blocked) of an individual actor and change in the topology of the system of actors.

First, sending a message from an active or root actor to a blocked acquaintance allows the blocked actor to become active. This transformation reflects the ability of an actor to alter the state (ready or blocked) of another actor.

The second transformation occurs when a root or active actor sends its own mail queue address or the mail queue address of one of its acquaintances to another of its acquaintances. This transformation reflects the ability of an actor to send a mail queue address, thereby changing the topology of the actor system.

Suppose an actor system has the above transformations repeatedly applied until no more transformations can be applied. Then, all actors which are not direct acquaintances of root actors in the resulting topology are garbage.

One key property of garbage actors is that they cannot become non-garbage. This is because actors are only determined to be garbage when there is no possibility of communication between it and a root actor. Therefore, once an actor is marked as garbage, there is no possible sequence of transformations which would cause the garbage actor to become non-garbage.

It is focal point of our thesis that how to retrieve garbage actor correctly. And then other definitions.

7. **Black Actor** : If the color of an actor is black, then it is a non-garbage. And, if an actor is a non-garbage, then its color is black.

8. **White Actor** : if an actor is a garbage, then its color is white.

9. **Visit-level** : In our algorithm, if $u \in V$, $v \in V$, $(u,v) \in E$.

   (a) u is an active actor.

   (b) v is not visited.

   We say that u and v are in the same visit-level. Later, we will show that actors in the same visit-level have the same color.

10. ***visit_level_number*** : It is an integer global variable which is used to assign each visit-level a unique number, and $0 \leq visit\_level\_number \leq n$.

11. ***Actor.color*** and ***Actor.vl*** :

    (a) *Actor.color*: It is an integer value, where

        i. *color* = 0 means that its color is black. And, the actor is non-garbage.

        ii. *color* = 1 means that its color is white.

    (b) *Actor.vl* : It is an integer value, where

        i. $vl = 0$: the actor is unvisited. And, the color of actor is white.

        ii. $vl = 1$: the actor is a non-garbage. And, the color of actor is black.

        iii. Otherwise: the actor has a visit-level vl, and $2 \leq vl \leq n$.

Then, we define data structures used in our algorithm.

1. ***Q***: Q is a set that contains all the unvisited active actors.

2. ***B***: B is a set that contains all black actors computed in the *LV_Initialization* phase. Note that each actor a in B has a visit level number of 1. ($\forall a \in B$, $a.vl$=1).

3. ***S***: S is a temporary set which collects the following actors:

   $S(u) = \bigcup_{i=0}^{n} S_i$ where

   $S_0(u) = \{v | (u,v) \in E,\ u$ is active.$\}$

   $S_i(u) = \{$where $\exists\ w \in S_{i-1}(u), (w,v) \in E,\ 0 < i \leq n\}$

4. ***P***: P is a set that contains all white actors in the *LV_Initialization* phase.

5. ***T***: T is an equivalence table which is used to record the relationships between visit-levels. T has the following attribute:

   - $(i,j) \in T \Rightarrow$The actors of visit-level i has the same color as the actors of visit-level j. $1 \leq j \leq$n,$1 \leq i \leq$n.

3.2 *Levelled_Visit* Algorithm

In this section we discuss the *Levelled_Visit* algorithm. The input of the algorithm is the actor reference graph of the source program, and the output of the algorithm is a set of actors garbage.

There are six co-algorithms used by the *Levelled_Visit* algorithm, including the *LV_Initialization* algorithm, the *Explore_Same_Level* algorithm, the *Update_Equivalence_Table* algorithm, the *Recursive_Is_Black* algorithm, the *Equivalent_to_Root* algorithm and the *Equivalent_to_Visit_Level* algorithm.

Figure 3.1 is the overall Structure of *Levelled_Visit* Algorithm.

Introduce these algorithms the following:

1. *LV_Initialization*: This algorithm is responsible for initialize data structure the used by the *Levelled_Visit* algorithm. And, all root actors and all the actors potentially receive a message from a root actor are assigned as visit-level 1, and black color. The other initialized as white actors.

2. *Explore_Same_Level*: This algorithm is responsible for choosing one active actors, and find out the actors of the same visit-level as the chosen actor. This algorithm also collects the visit-levels that the chosen actor

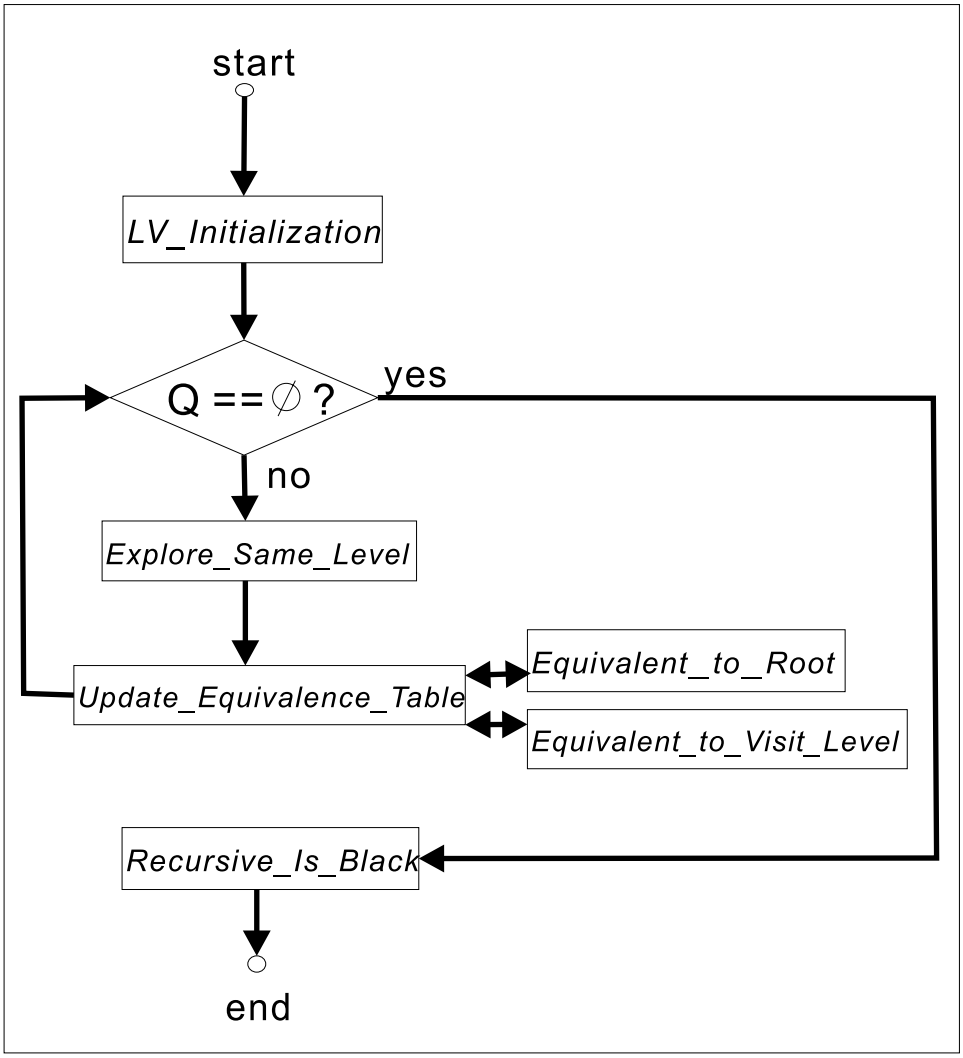

Figure 3.1: Overall Structure of *Levelled_Visit* Algorithm.

may reach into S.

3. *Update_Equivalence_Table*: This algorithm is responsible for updatly an equivalence table based on S.

4. *Equivalent_to_Root*: This algorithm is responsible for updatly the equivalence table when a visit-level potentially sends message to a root or potentially receives a message from a root actor.

5. *Equivalent_to_Visit_Level*: This algorithm is responsible for updatly element T when a visit-level can be reached from the visit-level of *visit_level_number*.

6. *Recursive_Is_Black*: This algorithm is responsible for computly whether an actor is garbage or not.

The *Levelled_Visit* algorithm is shown in Figure 3.2. The LV_Initialization algorithm is shown in Figure 3.3. The *Explore_Same_Level* algorithm is shown in Figure 3.4. The *Update_Equivalence_Table* algorithm is shown in Figure 3.5. The *Equivalent_to_Root* algorithm is shown in Figure 3.6. The *Equivalent_to_Visit_Level* algorithm is shown in Figure 3.7. The *Recursive_Is_Black* algorithm is shown in Figure 3.8.

Algorithm: *Levelled_Visit*

```
LV_Initialization;
while[Q!=⊘] DO
  Let a be an element in Q DO
    S:={visit_level_number};
    Explore_Same_Level(a);
  Update_Equivalence_Table;
  visit_level_number++;
END DO;

FORALL[a∈P]DO
  IF[Recursive_Is_Black(a.vl)]
    a.color := black;
  END IF;
END DO;
```

Figure 3.2: Algorithm: *Levelled_Visit* Algorithm

Algorithm: *LV_Initialization*

```
visit_level_number:= 0;
T with:= [(1,1)];
Q:= ⊘;
P:= ⊘;
S:= ⊘;
B:= ⊘;

FORALL[a is not a root actor] DO
   a.color := white;
   a.vl := visit_level_number;
   P with := a;
END DO;
visit_level_number:= 1;
FORALL[a is a root actor] DO
   a.color := black;
   a.vl := visit_level_number;
   B with := a;
END DO;
REPEAT
   flag := true;
   FORALL[b∈B]DO
      FORALL[bt is an acquaintance of b]DO
         IF[bt.vl==0]
            bt.vl := visit_level_number;
            bt.color := black;
            B with := bt;
            P less := bt;
            flag := false;
         END IF;
      END DO;
   END DO;
UNTIL[flag];
FORALL[a∈P]DO
   IF[a is active]
      Q := Q with a;
   END IF;
END DO;
visit_level_number := 2;
```

Figure 3.3: Algorithm: *LV_Initialization* Algorithm

Algorithm: *Explore_Same_Level*

```
input: actor
  IF[actor.vl==1] //found a black actor
    S with := 1;
  ELSE IF[actor.vl==0] //actor has not been visited before
    actor.vl := visit_level_number;
    IF[actor is active]
      Q less := actor;
    END IF;
    FORALL[at is an acquaintance of actor]DO
      Explore_Same_Level(at);
    END DO;
  ELSE IF[actor.vl != visit_level_number]
    //found actor has been visited before
    S with := actor.vl;
  //ELSE[actor.vl==visit_level_number] do nothing;
  END IF;
```

Figure 3.4: Algorithm: *Explore_Same_Level* Algorithm

Algorithm: *Update_Equivalence_Table*

```
T with := (visit_level_number,visit_level_number);
IF[1∈S]
  S less := 1;
  FORALL[a∈S]DO
    Equivalent_to_Root(a);
  END DO;
ELSE
  FORALL[a∈S]
    Equivalent_to_Visit_Level(a);
  END DO;
END IF;
```

Figure 3.5: Algorithm: *Update_Equivalence_Table* Algorithm

Algorithm: *Equivalent_to_Root*

input: a
IF[$T[a]$==a]
$T[a] := 1$;
ELSE IF[$T[a]$!=1]
$b := T[a]$;
Equivalent_to_Root(b);
$T[a] := 1$;
//ELSE IF[$T[a]$==1]do nothing;
END IF;

Figure 3.6: Algorithm: *Equivalent_to_Root* Algorithm

Algorithm: *Equivalent_to_Visit_Level*

input: a
output: boolean

IF[$T[a]$==1]
T[*visit_level_number*] := 1;
return true;
ELSE IF[$T[a]$==a]
$T[a] :=$ *visit_level_number*;
return false;
ELSE // $T[a]$!= a
$b := T[a]$;
IF[*Equivalent_to_Visit_Level(b)*==true]
$T[a] := 1$;
return true;
ELSE
$T[a] :=$ *visit_level_number*;
return false;
END IF;
END IF;

Figure 3.7: Algorithm: *Equivalent_to_Visit_Level* Algorithm

Algorithm: *Recursive_Is_Black*

input: *vl*
output: boolean

```
IF[vl==0]
  //The actors of visit level 0 are blocked
  //and have not been visited in the while loop.
  return false;
END IF;
IF[T[vl]==1]
  return true;
ELSE IF[T[vl]!=vl]
  v = T[vl];
  b = Recursive_Is_Black(v);
  IF[b]
    T[vl] := 1;
  ELSE
    T[vl] := vl;
  END IF;
  return b;
ELSE //T[vl]==vl;
  return false;
END IF;
```

Figure 3.8: Algorithm: *Recursive_Is_Black* Algorithm

3.3 An Example

In this section, we use the example in Figure 3.9 to demonstrate how the *Levelled_Visit* algorithm finds the garbage actors.

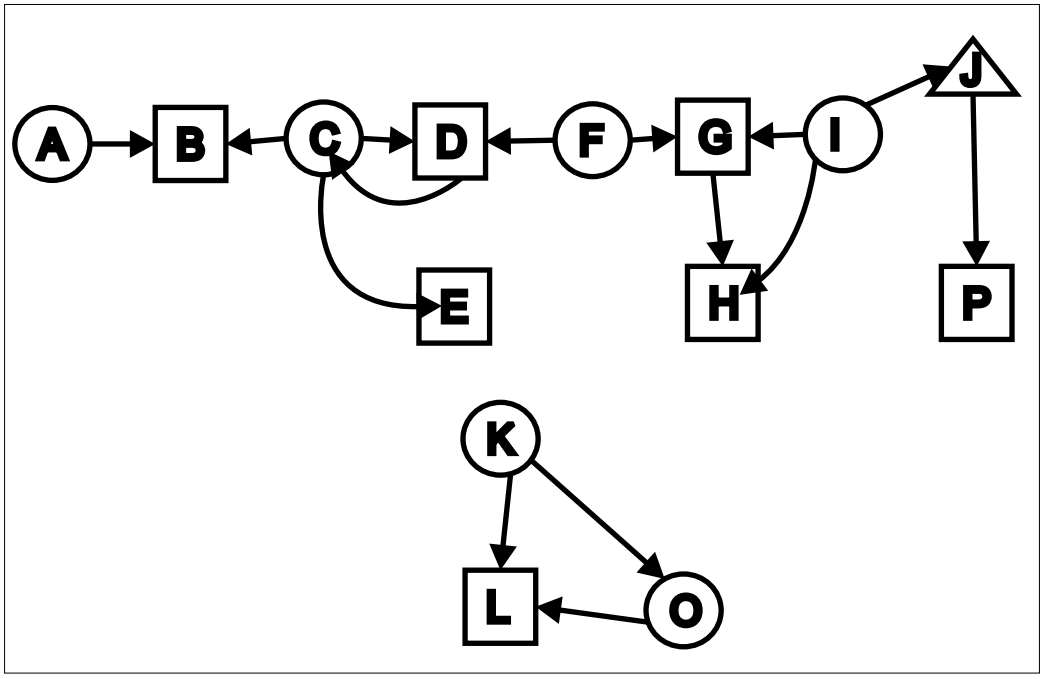

Figure 3.9: Using Levelled_Visit Algorithm to Collect Garbages of Actor Systems.

The *LV_initialization* algorithm puts actors J and P into the $\boldsymbol{B}$ set (i.e. color of J and P are black, and their vl are set to 1). Then, it put actors A,B,C,D,E,F,G,H,I,K,L and P into the $\boldsymbol{P}$ set. Next, it puts actors A,C,F,I,K and O into the $\boldsymbol{Q}$ set. When the *LV_Initialization* algorithm finishes, $visit\_level\_number = 2$, and, T=[(1,1)].

For simplicity, we assume that actors are examined in alphabetical order. In the first iteration, because $\boldsymbol{Q}$ set is not empty, we pick actor A form $\boldsymbol{Q}$ to do *Explore_Same_Level algorithm*. When the *Explore_Same_Level algorithm* is

finished, $\boldsymbol{S}$ ={2}, actors A and B have the same visit-level of 2. When the *Update_Equivalence_Table* algorithm is finished, T =[(1,1),(2,2)]. Now the $\boldsymbol{Q}$ set has actors C,F,I,K and O, and *visit_level_number* = 3.

In the second iteration, because $\boldsymbol{Q}$ set is not empty, we pick actor C form $\boldsymbol{Q}$ to do *Explore_Same_Level algorithm*. When the *Explore_Same_Level algorithm* is finished, $\boldsymbol{S}$ ={2,3}, actors C and D have the same visit-level of 3. When the *Update_Equivalence_Table* algorithm is finished, T =[(1,1),(2,3),(3,3)]. So we know visit-level 2 has the same color as the visit-level 3. Now the $\boldsymbol{Q}$ set has actors F,I,K and O, and *visit_level_number* = 4.

In the third iteration, because $\boldsymbol{Q}$ set is not empty, we pick actor F form $\boldsymbol{Q}$ to do *Explore_Same_Level algorithm*. When the *Explore_Same_Level algorithm* is finished, $\boldsymbol{S}$ ={3,4}, actors F,G and H have the same visit-level of 4. When the *Update_Equivalence_Table* algorithm is finished, T =[(1,1),(2,3),(3,4),(4,4)]. So we know visit-level 3 has the same color as the visit-level 4. Now the $\boldsymbol{Q}$ set has actors I,K and O, and *visit_level_number* = 5.

In the forth iteration, because $\boldsymbol{Q}$ set is not empty, we pick actor I form $\boldsymbol{Q}$ to do *Explore_Same_Level algorithm*. When the *Explore_Same_Level algorithm* is finished, $\boldsymbol{S}$ ={1,4,5}, actor I has the visit-level of 3. When the *Update_Equivalence_Table* algorithm is finished, T =[(1,1),(2,3),(3,4),(4,1),(5,1)]. So we know visit-level 4 and visit-level 5 have the same color as the visit-level 1. Now the $\boldsymbol{Q}$ set has actors K and O, and *visit_level_number* = 6.

In the fifth iteration, because $\boldsymbol{Q}$ set is not empty, we pick actor K form $\boldsymbol{Q}$ to do *Explore_Same_Level algorithm*. When the *Explore_Same_Level algorithm* is

finished, $\boldsymbol{S}$ ={6}, actors K,L and O have the same visit-level 6. When the *Update_Equivalence_Table* algorithm is finished, T =[(1,1),(2,3),(3,4),(4,1),(5,1),(6,6)]. So we know visit-level 4 and visit-level 5 have the same color as the visit-level 1. Now $\boldsymbol{Q}$ set is empty, and *visit_level_number* = 7.

Now $\boldsymbol{Q}$ is empty, and *Recursive_Is_Black* algorithm start. First, the *Recursive_Is_Black* algorithm computes for the actor A in $\boldsymbol{P}$, which return true. And, T = [(1,1),(2,1),(3,1),(4,1),(5,1),(6,6)] at this moment.

Next, the *Recursive_Is_Black* algorithm computes for the actor B in $\boldsymbol{P}$, which returns true.

In a similar way,the *Recursive_Is_Black* algorithm computes for the actors C,D,E,F,G,H and I, respectively. All computation return true.

Next, the *Recursive_Is_Black* algorithm computes for the actor K in $\boldsymbol{P}$, which returns false.

In a similar way,the *Recursive_Is_Black* algorithm computes for the actors L and O, respectively. All computation return false.

Finally, we can see the actors A,B,C,D,E,F,G,H and I are non-garbage. The actors K,L and O are garbage.

Chapter 4

Analysis of *Levelled_Visit* Algorithm

In this chapter, we analyze the correctness and computational complexity of our algorithm.

4.1 Proof of Correctness

In this section, we first give the lemmas. Proving the theorem finally.

Lemma 1: $\exists(u,v) \in E$, *u is an active actor, then*

1. *if u is black then v is black.*

2. *if v is black then u is black.*

That is, u and v are of the same color.

Proof:

1. If u is black, then u potentially sends a message to a root, or u potentially

receives a message from a root. And, if $(u,v) \in E$, by the above, we can know v potentially sends a message to a root, or v potentially receives a message from a root. And hence, v is black.

2. If v is black, then v potentially sends a message to a root, or v potentially receives a message from root. And, if $(u,v) \in E$, and u is active, by the above, we can know u potentially sends a message to a root, or u potentially receives a message from root. And hence, u is black.

By 1 and 2, we complete the proof.□

Lemma 2: *The actors in a visit-level have the same color of each other. That is,* $\forall a \in V, \forall b \in V, a.vl = b.vl \Leftrightarrow$ *a and b have the same color.*

Proof: By the definition of visit-level, $\forall a,b.\ a \in V,\ b \in V,\ a.vl = b.vl$. One and only one of the following three cases holds.

1. $(a,b) \in E$, a is active. By Lemma 1, a and b have the same color.

2. $(b,a) \in$ E, b is active. By Lemma 1, b and a have the same color.

3. $\exists$ an actor $c \in V$, c is active, $(c,a) \in E$, $(c,b) \in E$. By Lemma 1, c and a have the same color. By Lemma 1, c and b have the same color. Hence, a and b have the same color.

This completes the proof.□

Lemma 3: *If* $(u,v) \in E$, *u is active and* $u.vl \neq v.vl$, *then the actors in visit-level u.vl have the same color as the actors in visit-level v.vl.*

Proof:

1. Since u is active and $(u,v)\in E$, by lemma 1, u and v have the same color.

2. By lemma 2, the actors in visit-level $u.vl$ have the same color as u.

3. By lemma 2, the actors in visit-level $v.vl$ have the same color as v.

By the above, we know the actors in visit-level $u.vl$ have the same color as the actors in visit-level $v.vl$. This completes the proof.□

Lemma 4: *After the computation of the Levelled_Visited algorithm, if $k=T[i]$, the actors of visit-level i have the same color as the actors in visit-level k.*

Proof:

1. First, we define U: $U = [(i,j)|i\neq j,u\in V,v\in V,i=u.vl,\ j=v.vl,\ v$ is active, $(v,u)\in E]$. $u[i]=j$ means that actors of visit-level i have the same color as the actors in visit-level j(by lemma 3).

2. It is clear that after the computation of the *Levelled_Visit* algorithm, $T[i]=k\Leftrightarrow \exists$ a sequence $l_1,...,l_m$. $m<n$. Such that $u[i]=l_1, u[l_1]=l_2,...,u[l_{m-1}]=l_m, u[l_m]=k$.

Thus, it is clear that the actors of visit-level i have the same color as the actors in visit-level k.□

Lemma 5: *All the non-garbage actors are black after the computation of the Levelled_Visit algorithm.*

Proof: if actor u is non-garbage⇒ actor u potentially send or receive to/form a root actor.⇒ $T[u.vl] = 1$.⇒ By the computation of *Recursive_Is_Black* algorithm, u is black after the computation of the *Levelled_Visit* algorithm. This completes the proof.□

Theorem: *All the white actors computed by the Levelled_Visit algorithm are garbage.*

Proof: It is clear that the theorem is a corollary of Lemma 5.□

4.2 Computational Complexity

In this section, we compare the worst-case space and time complexity of the *Levelled_Visit* algorithm in the pervious section.

4.2.1 Space Complexity

There is a recursive call in the *Explore_Same_Level* algorithm, which require stack space. Each recursion requires a constant amount of stack space, and the worst-case number of recursion is the maximum of:

1. Number of actors explored until a cycle is reached.
2. Number of actors explored until an actor with no acquaintances is reached.

So this Space Complexity = $k_1 N$.

Where N is the number of actor in the system and k_1 is the size of a stack frame on recursion.

In addition, if there are N white active actors, and T has N elements, then space complexity of T is *2N*.

Thus, the worst-case space complexity is:

$$\text{Space Complexity} = k_1\text{N} + 2N = \text{O}(N);$$

4.2.2 Time Complexity

There are six co-algorithm used by the*Levelled_Visit* algorithm. We discuss each algorithm individually.

1. *LV_Initialization algorithm*: The algorithm must be computed all actors in actor reference graph, so it is Time Complexity = O(N). Where N is the number of actor in the system.

2. *Explore_Same_Level algorithm*: The algorithm has recursively run itself until where:

   (a) Number of actors explored until a cycle is reached.

   (b) Number of actors explored until an actor with no acquaintances is reached.

   So we know the algorithm Time Complexity = O($|E| + |N|$). Where N is the number of actor in the system, and E is the number the edges of actor in the system.

3. *Update_Equivalence_Table* algorithm: The algorithm has two issues:

(a) Update an element of T is constant cost. So Time Complexity is O(1).

(b) If there are N white active actors, and T has N elements, then *Levelled_Visted* algorithm is running, each $T[i]$, i= 1 to N, $T[i]$ is replaced N times by k from j, $(j,k)\in E$. So the worst case time complexity is:

Complexity = (element number of T) * Each element is replaced N times.

Complexity = O(N)*O(N)=O(N^2)

It is total cost for updating T is O(N^2).

4. *Recursive_Is_Black* algorithm: If T has N elements, the algorithm determines the actor whether it is garbage or not, must recursively run itself N times. The Time Complexity is O(N^2).

5. *Equivalent_to_Root*: If T has N elements, the algorithm updates the elements of T, and recursively runs itself N times. The Time Complexity is O(N).

6. *Equivalent_to_Visit_Level* algorithm: If T has N elements, the algorithm updates the elements of T, and recursively runs itself N times. The Time Complexity is O(N).

So we know the *Levelled_Visit* algorithm which is Time Complexity is : O(N) + O($|E|+|N|$) +O(N^2) + O(N^2) + O(N) + O(N) = O(N^2).

Chapter 5

Experiment and Result

This chapter describes the experiment of garbage collection of actors in a embedded environment.

5.1 Underlying architecture

We have implemented a garbage collector for actors in the our GCofActorsSimulator, which is an Actor system simulator written in $Java^{TM}$. Our GCofActorsSimulator will apply to J2ME[JA05] application. $Java^{TM}$ Platform, Micro Edition (Java ME) is the most ubiquitous application platform for mobile devices across the globe. It provides a robust, flexible environment for applications running on a broad range of other embedded devices, such as mobile phones, PDAs, TV set-top boxes, and printers.

We use the Netbeans IDE[NE05](Integrated Development Environment) to develop our GCofActorsSimulator. The NetBeans Mobility Packs can be used to write, test, and debug applications for the Java Micro Edition platform

(Java ME platform) technology-enabled mobile devices. The NetBeans Mobility Pack integrates support for the Mobile Information Device Profile (MIDP) 2.0, the Connected, Limited Device Configuration (CLDC) 1.1. We can easily integrate third-party emulators for a robust testing environment.

GCofActorsSimulator offers two kinds of functions mainly where:

1. Implement three kinds of garbage collection algorithms, where:
   (a) *Push-Pull* Algorithm,
   (b) *Is_Black* Algorithm, and
   (c) *Levelled_Visit* algorithm.
2. Offer Actor reference model to garbage collect for garbage collection algorithm.

In next section, we will discuss implementation of GCofActorsSimulator in detail.

5.2 Implementation

For the implementation of the GCofActorsSimulator, we have provided a Java class *JActorRetrieve* to implement the algorithm of garbage collection. It is the super class. Three classes will inherit this super class. Including *JPush-PullRetrieve* class, *JIsBlackRetrieve* class and *JLevelledVisitRetrieve* class.

We have also provided a Java class *JActorReferenceModel* to implement the Actor reference model. This class offers Actor reference model for class *JActorRetrieve* use.

There are two kinds of *JActorReferenceModel* ways of producing Actor reference model, one is actually built and constructed Actor reference model according to the script, another one produces actor reference model to at random, which has amount of actor between 300 and 10000.

We have measured the simulator of collectors on the platform: a Pentium based ASUS AB-P2300. CPU is Pentium-4,which clock time is $3GH_z$. RAM is 1GB and OS is Windows XP professional which version is 2002.

In next section, we will give some test cases for GCofActorsSimulator, and also give test results.

5.3 The Result

In this section, we show the experimental result for a set of test cases and its detail as shown in Figure 5.1.

Test Case	# of actors	# of root actors	# of active actors	# of blocked actors	# of edges
A	302	2	140	160	538
B	505	5	200	300	539
C	1160	10	400	750	1950
D	2215	15	750	1450	3780
E	4030	30	1200	2800	6190
F	5040	40	2000	3000	5560
G	6060	60	2500	3500	6700
H	8070	70	3000	5000	8880
I	9080	80	4000	5000	10174
J	10000	90	4010	5900	11982

Figure 5.1: The Test Data

In each test case, we repeat each test case 100 times, then averaged total collection time. Edges of each test case is random associate any actors. We let

user appoint number of root, active and locked actors. And number of edge is define by user.

The result in Figure 5.2.

In these test cases, the execution time of garbage collection is reduced from 2.128% to 64.98% by our algorithm over the Push-Pull(33.191% $\sim$ 64.98%, averaged at 49.924%) and Is-Black(2.128% $\sim$ 25.129%, averaged at 12.92%) algorithms.

Test Case	Levelled_Visit(LV)	Push-Pull (PP)		Is_Black (IB)	
Name	LV run time(t1)	PP run time(t2)	(t2-t1)/t2	IB run time(t3)	(t3-t1)/t3
A	0.92ms	1.6ms	42.500%	0.94ms	2.128%
B	1.57ms	3.44ms	54.360%	1.9ms	17.368%
C	5.19ms	14.82ms	64.980%	5.81ms	10.671%
D	15.62ms	23.38ms	33.191%	16.24ms	3.818%
E	39ms	64.48ms	39.516%	50.62ms	22.955%
F	37.6ms	92.86ms	59.509%	50.22ms	25.129%
G	51.32ms	132.48ms	61.262%	59.4ms	13.603%
H	73.44ms	186.24ms	60.567%	91.94ms	20.122%
I	109.26ms	188ms	41.883%	118.38ms	7.704%
J	149.64ms	255.66ms	41.469%	158.68ms	5.697%

Figure 5.2: The result

Chapter 6

Conclusion

In this thesis, we introduce the related techniques about actor model and garbage collection of Actors. Based on the garbage of actors defined by Nelson, we propose the Levelled-Visit algorithm to collect garbage. We define the concept of visit-level, in which all actors have the same color. Improved from the Push-Pull algorithm and the Is-Black algorithm for garbage collection proposed by Kafura et al., our algorithm eliminate the repeated visits to actors. This thesis also includes the proof of correctness and the complexity analysis. The time complexity of our algorithm is O(N^2), which the same as the Push-Pull and Is-Black algorithms. We have implemented our algorithm in the J2ME. J2ME is the most ubiquitous application platform for mobile embedded devices. We experiment on the time required by various garbage collection algorithms, which have Push-Pull, Is_Black and Levelled_Visit algorithms. The execution time of garbage collection is reduced from 2.128% to 64.98% by our algorithm over the Push-Pull(33.191% $\sim$ 64.98%) and Is-Black(2.128% $\sim$

25.129%) algorithms.

Bibliography

[BA04] David F.Bacon, Perry Cheng and David Grove, "Garbage Collection for Embedded Systems". September 2004 Proceedings of the 4th ACM international conference on Embedded software EMSOFT '04

[CU00] CULLER, D. E., HILL, J., BUONADONNA, P., SZEWCZYK, R., AND WOO, A. A network-centric approach to embedded software for tiny devices.

[CO60] COLLINS, G. E. A method for overlapping and erasure of lists. Commun. ACM 3, 12 (Dec. 1960), 655V657.

[CJ70] CHENEY, C. J. A nonrecursive list compacting algorithm. Commun. ACM 13, 11 (1970), 677-678.

[DK90] Dennis kafura and keung Lee, "ACT++: Building A Concurrent C++ With Actors," Jounral of Object-Oriented Programming, to appear, 1990, also Technical Report 89-18, Department of Computer Science, Virginia Tech, Blacksburg, VA.

[FY69] FENICHEL, R. R., AND YOCHELSON, J. C. A LISP garbage collec-

tor for virtual-memory computer systems. Commun. ACM 12, 11 (Nov. 1969), 611V612.

[GA86] G. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Gambridge, Mass., 1986

[GN99] G. Agha and N. Jamali. Concurrent programming for distrubuted artifical intelligence. In G. Weiss, editor, *Multiagent Systems: A modern Approach to DAI.*, chapter 12. MIT Press, 1999.

[JA05] http://java.sun.com/javame/index.jsp

[JO96] JONES, R., AND LINS, R. Garbage Collection. JohnWiley and Sons, 1996.

[KWN90] Dennis Kafura, Doug Washabaugh, and Jeff Nelson. Garbage collection of actors. In Norman Meyrowitz, editor. OOPSLA90 ACM Conference on Object-Oriented Systems, Languages and Applications, volume25(10) of ACM SIGPLAN Notices, pages 126-134, Ottawa, Ontario, October 1990 ACM Press.

[MJ60] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. Commun. ACM 3, 4 (1960), 184V195.

[NE05] http://www.netbeans.org/index.html

[NE89] Jeff Nelson, Automatic, Incremental, On-the-fly Garbage Collection of Actors, M.S Thesis, Department of Computer Sceince, Virginia Tech, Blacksburg, VA, February 1989.

[NE89] Jeffery E. Nelson. Automatic, incermental, on-the-fly garbage collection of actors. Master's thesis, Virginia Polytechnic Institute and State University, 1989.

[WI03] WILSON, G., OSTREM, J., AND BEY, C. Palm OS Programmers Companion, vol. 1. Palm Source, Inc., 2003. Document no. 3004-008.

[WG95] W. Kim and G. Agha. Eficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Proceedings of Supercomputing'95*, 1995.

[YA00] http://yangtze.cs.uiuc.edu/foundry/