COARSE GRAIN MEMORY MANAGEMENT IN REAL-TIME SYSTEMS

By
Andrew Borg

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT THE
UNIVERSITY OF YORK
YORK, UK
MARCH 2006

To my Family and Nesha

Abstract

As real-time and embedded systems become increasingly large and complex, the traditional strictly static approach to memory management begins to prove untenable. The high development costs and space requirements of the static approach need to be reduced at a minimal performance penalty and without compromising the predictability of the system. Therefore, the challenge is to provide a dynamic memory model that guarantees tight and bounded time and space requirements without overburdening the developer with memory concerns.

The Real-Time Specification for Java (RTSJ) introduces a memory model in which applications create objects in scoped memory regions that can be allocated and freed dynamically. This allows applications to avoid garbage collection latencies and greatly reduces the fragmentation problems that a fine grain explicit memory management solution would incur. However, the RTSJ provides little information on how the model would be implemented in a virtual machine. In particular, there is no explicit requirement on the predictability of region allocation and deallocation. A second shortcoming – the high complexity of using the model in real-world applications – has inhibited uptake of the RTSJ.

This thesis argues that a dynamic, coarse grain approach to memory management can help deliver tight time and space overheads at an acceptable development cost. The rationale behind this argument is formulated in a simple hypothesis that describes the correlation between space overheads, time overheads and program semantic information expressible and usable in a memory model. Adopting a scoped approach to coarse grain memory management such as that provided in the RTSJ will be shown to derive a tradeoff between space and time overheads that is attractive at times, but less so at others.

The key contributions described in this thesis are constructed on a model that allows the lifetime of coarse grain memory regions to be expressed. Called the Region Partitioning Model, it enforces a separation between memory concerns and program logic and thereby serves two purposes. Firstly, it allows an exploration into the repercussions of enforcing a scoped approach to coarse grain memory management as well as the RTSJ's implementation of this model. This motivates a number of changes and extension that address the shortcomings of the memory model as currently presented in the RTSJ. Secondly, by separating the memory management aspect from the application, it provides the basis for an environment offering low development and maintenance costs. The result is a model for developing RTSJ applications and a complementing extended RTSJ that addresses the prohibitive complexity of current development approaches while maintaining the performance and predictability advantages of the coarse grain model.

Table of Contents

List of Tables

List of Figures

Acknowledgements

This study would not have been possible were it not for the guidance and assistance of my supervisor, Professor Andy Wellings. His direction transformed my ideas into this final format. The result is a testimony to his professional tutorship. I am greatly indebted to him for his time, patience, understanding and support.

I would also like to thanks Professor Chris Gill, Professor Ron Cytron and Dr. Angelo Corsaro who made me so welcome at the University of Washington in St. Louis during my three month visit. The time spent there planted the key ideas from which this thesis developed.

My thanks go to friends and colleagues in York, in particular Professor Alan Burns for his helpful comments in the assessment process. I would also like to thank Dr. Neil Audsley, Dr. Guillem Bernat, Dr. Iain Bate, Dr. Ian Broster, Dr. Hao Cai, Dr. Rui Gao, Mr. Ameet Patil, Dr. Michael Ward and Mr. Mark Bartlett. I also thank former colleagues at the University of Malta who guided me in those first important steps towards becoming a researcher. In particular I would like to thank my former supervisor Dr. Kevin Vella as well as Mr. Kurt Debattista and Mr. Joe Cordina. I would like to thank my friends in Malta who were so instrumental in making special those all important breaks. In particular I would like to thank my closest friends Michael, Wallace, Mark and Robert. Special thanks go to Joe and his family who have been so generous with me in their time and support.

I would also like to thank my closest relatives, particularly Aunt Tania for always being so helpful with any travel needs, Aunt Rose and cousin Claire for keeping my sugar intake at suitably high levels and Uncle Michael who always goes to extraordinary lengths to obtain anything he might have heard I wished. Special mention must go to my grandmother and Aunt Domenica who I know always kept me in their prayers.

Finally, to my parents, my brother Matthew and his wife Ruth, and my wonderful girlfriend Nesha, my gratitude for their patience, love and support. This work has been as much an effort on their part as it was on mine.

York, UK
March, 2006

Andrew Borg

Declaration

I hereby declare that, unless otherwise indicated in the text, the research presented in this thesis is original work, undertaken by myself, Andrew Borg, between October 2001 and March 2006. I have acknowledged external sources through bibliographic referencing. Some parts of this thesis have appeared previously in the papers listed below. In all cases, the major contributions were made by myself with input from my supervisor, Andy Wellings.

The work in Chapters 4 and 5 is an extension of work first published as a technical report at the University of York [33] and subsequently in a paper presented at the Java Technologies for Real-Time and Embedded Systems (JTRES) workshop in 2004 [36]. A consolidation of this work is to appear in the International Journal of Embedded Systems in mid-2006 [38]. Elements of the work in Chapter 3 are also described in these three papers and are developed in further detail in a paper to be presented at the 2006 Euromicro Conference on Real-Time Systems [37] and as a technical report co-published by the University of York and the University of Washington in St. Louis [39].

The work on scoped reference objects described in Chapter 6 is based on work presented at the JTRES workshop in 2003 [35]. The ideas described in Chapter 8 with regards to the extension of the Region Partitioning Model to describe object lifetimes in distributed, real-time environments were also first proposed in this paper as an extension to the Author's work on real-time RMI. This latter work is described in a paper that was presented at the 2003 Euromicro Conference on Real-Time Systems [34] and subsequently published as an extended version in a University of York technical report [32].

List of Abbreviations

Abbreviation	Meaning
DMA	Dynamic Memory Allocator; an algorithm that allocates and deallocates memory during time-critical execution
LSL	Lifetime Specification Language; A language for describing information on memory usage. This is one of the two components of a memory model (the other being the memory subsystem)
STS	Spatio-Temporal Specification; A specification of memory usage by an application using the notation of a chosen LSL
RPM	Region Partitioning Model; used to encompass any of the LSLs in the region-based, coarse grain model
$\text{RPM}^{\rightarrow}$	The Region Partitioning Model LSL with only sequential flows in a single-threaded environment
$\text{RPM}^{?}$	The Region Partitioning Model LSL with only sequential flows and conditional branching in a single-threaded environment
$\text{RPM}^{\circlearrowleft}$	The Region Partitioning Model LSL with only sequential and iterative flows in a single-threaded environment
$\text{RPM}^{\parallel}$	The Region Partitioning Model LSL with only sequential flows in a multi-threaded environment
$\text{RPM}^{\parallel\sigma}$	$\text{RPM}^{\parallel}$ augmented with the ability to express region sharing across threads
$\text{RPM}^{\parallel\sigma\rho}$	$\text{RPM}^{\parallel\sigma}$ augmented with the ability to express region synchronisation between threads
eRPM	The Constrained Region Partitioning Model LSL with sequential and iterative flows and conditional branches in a single-threaded environment
$\text{eRPM}^{\parallel}$	The multi-threaded version of eRPM
$\text{eRPM}^{\maltese}$	A version of eRPM that simplifies the specification of lifetime information at the cost of additional space overheads
$\text{eRPM}^{\parallel\sigma\rho}$	The Region Partitioning Model LSL with all control flows in a multi-threaded environment and which also supports region sharing and synchronisation

Abbreviation	Meaning
cX	The constrained versions of the above LSLs; for example cRPM$^{\rightarrow}$ is the constrained version of RPM$^{\rightarrow}$
τ_j	Thread j of a program in an LSL that supports multi-threading
r_i^j	A region r_i that is a member of the regions of thread τ_j
LAP(r_i^j)	The Latest Allocation Point of a region; in general this is just before the execution of that region in the same thread
RAP(r_i^j)	The Real Allocation Point of a region; this is the allocation point in constrained versions of the RPM and lists locations at which this region can be allocated in the control flow of τ_j
EDP(r_i^j)	The Earliest Deallocation Point of a region; this is the earliest point at which memory for a region can be deallocated; the elements of this set will list the earliest points that the memory for this region can be deallocated for every thread in the program
RDP(r_i^j)	The Real Deallocation Point of a region; similar to EDP(r_i^j), this is the point at which memory for a region can be deallocated but holds for the constrained versions of the RPM
$\widetilde{\text{RDP}}(r_i^j)$	The subset of deallocation points in RDP(r_i^j) that specifies the deallocation point of region r_i^j in threads other than τ_j; used to simplify the notation used to specify deallocation points
EMM − ONLINE	The Explicit Memory Model memory subsystem for unconstrained versions of the RPM that do not take advantage of knowledge of the lifetime of regions
EMM − OFFLINE	The Explicit Memory Model memory subsystem for unconstrained versions of the RPM that take advantage of knowledge of the lifetime of regions by employing compile-time analysis
SMM$^{\rightarrow}$	The Scoped Memory Model memory subsystem; the memory subsystem for constrained versions of the RPM that take advantage of region ordering to derive a scoped order
SMM$^{\parallel\sigma}$	The Scoped Memory Model memory subsystem for multi-threaded systems with specified region sharing
SMM$^{\parallel\sigma\rho}$	The Scoped Memory Model memory subsystem for multi-threaded systems with specified region sharing and region synchronisation
$\mathcal{M}_{MS}^{LSL}$	A memory model with name $\mathcal{M}$, Lifetime Specification Language LSL and memory subsystem MS

Abbreviation	Meaning
$F^{\mathcal{M}_{MS}^{LSL}}$	Function F as applied to the memory model using the Lifetime Specification Language LSL and memory subsystem MS; for example $\text{MemReq}^{\mathcal{M}_{MS}^{LSL}}(P)$ is the memory requirements of P that targets memory model $\mathcal{M}$ using Lifetime Specification Language LSL and memory subsystem MS
RPM − Unscoped	The general name for memory models that use unconstrained versions of the RPM as a LSL
RPM − Scoped	The general name for memory models that use constrained versions of the RPM as a LSL and where these constraints are such that region allocation and deallocation is in scoped order

Chapter 1

Introduction

Memory management is a major concern when developing real-time and embedded applications. Indeed, the cost of memory is often the single most expensive component of these systems [97]. The unpredictability of memory allocation and deallocation has resulted in hard real-time systems being necessarily static. Even when the analytical worst case space and time requirements can be derived [105, 100], the overheads of using dynamic memory are too large. Increasingly however, the complexity of today's real-time systems is making a purely static use of memory untenable; preallocated memory incurs a high space overhead and circumventing this through complex reuse techniques results in a high development cost. Real-time developers require a memory management solution that can provide a simple way of expressing dynamic memory usage in applications whilst minimising average and, more importantly, worst case space and time overheads.

As a language which by design frees the user from memory concerns, Java has proven attractive for the development of highly dynamic applications. In fact, the Java Programming Language Specification [69] describes no mechanism for reclaiming the memory used by unreachable objects. Consequently, an implementation must provide some form of garbage collection. However, real-time threads may fail to meet their timing requirements if interrupted by the garbage collector. The real-time community is now looking at Java as a solution for programming dynamic real-time applications [68, 31, 115] and several solutions that make memory management in Java suitable for real-time environments are being proposed. One approach is to adopt incremental garbage collectors [16, 14] as these provide bounds on the latencies a thread may experience due to garbage collection. Nevertheless, one commonly finds a lack of confidence in these algorithms in time-critical applications as they exhibit time and space overheads that are deemed too high. The Real-Time Specification for Java (RTSJ) [68] proposes an alternative in the form of a unique memory model based on scoped memory regions [118]. However, this model is often criticised for being inflexible and complex to use.

Achieving a memory model that can provide predictability *and* low space and time overheads

is a key objective of the Java and real-time and embedded systems communities. The arrival of the RTSJ has resulted in a further injection of research into this area. There are two questions of interest to the field:

(Q1): What is the best memory model to use in a particular class of real-time systems?

(Q2): What is the best memory model to use given the syntax and semantics of the Java Language and Virtual Machine Specifications?

The answers to these two questions are not necessarily identical. (Q1) seeks to identify the ideal memory model for a class of real-time systems and then build the language and programming paradigm around it. Answering the second question requires consideration of Java's programming model and may require modifications or additions to this model. For example, the safety afforded by Java through its referential integrity guarantees would ideally be maintained. The solution to (Q2) could therefore either be a model totally distinct to the solution for (Q1) or a strategy of implementing the latter within a Java environment.

Identifying and characterising the criteria used to gauge the advantages and disadvantages of different memory models is non-trivial. Arguing that one memory model is better than another must be qualified by a specification of the driving design forces. For example, although real-time garbage collection may be suitable for some types of applications, it will be shown that the approaches adopted in existing implementations fail to fill the gap between the requirements of smaller real-time applications for which a static approach is suitable and more complex systems for which space and time requirements are still tightly bounded. One important element of these driving forces is the application development cost of the chosen model with non real-time garbage collection being the best choice in this regard as the development cost of using memory dynamically is minimal. Real-time garbage collectors introduce an extra level of complexity as developers must specify necessary parameters in order to meet a more stringent set of guarantees. These parameters are often not easy to capture. The real-time community, particularly advocates of real-time Java technologies, remain highly polarised as to whether real-time garbage collection can ever achieve the predictability and efficiency required by these systems. Critics of garbage collection technologies argue that the space-time tradeoffs achievable by real-time garbage collectors are still unsuitable for the class of systems targeted by the static paradigm. In order to derive the required performance in one dimension, the costs in the other are too high. On the other hand, critics of new memory models such as that proposed in the RTSJ argue that the burden of memory concerns placed on developers detracts from what is arguably the most attractive feature of developing large and complex systems in environments such as Java: freedom from memory concerns. The common ground between these two camps is that the success or failure of providing a suitable environment for developing the next generation of

complex real-time and embedded systems hinges on getting the memory model right.

1.1 Towards Dynamic Memory Management in Real-Time Systems

Memory management has been the subject of extensive research for several decades. As a limited and comparatively expensive resource that is competed for by tasks within increasingly large and complex applications, its use must be efficient in both space and time. Two functions of a memory management mechanism must be considered:

1. application requests for memory reservation and subsequent freeing of that memory (allocation and deallocation) and

2. access to this reserved memory.

Modern computer architectures employ complex mechanisms such as multi-level caches, interleaving, bursting and pipelining for fast memory access. Space and time efficiency in servicing requests for memory allocation and subsequent deallocation depends on the strategy employed in selecting the area of memory to provide when a request is made. The algorithm responsible for executing this strategy is the Dynamic Memory Allocator (DMA). In particular, the DMA needs to consider memory fragmentation that may result in higher space and time costs, failed allocation requests or longer response times to service allocation requests.

In a real-time environment, efficiency comes second to predictability. In particular, the average time for servicing an allocation request or executing a memory access is of less importance than the analytical worst case. However, the various types of real-time systems — from hard, safety-critical systems to softer, Quality-Of-Service type systems — place differing degrees of emphasis on this argument. This thesis is concerned with *dynamic memory allocation and deallocation in hard real-time systems*. A *dynamic system* is defined as a system in which memory may be reclaimed and reused while the application is executing. A *hard real-time system* is defined as one in which executing tasks must meet a stipulated deadline or the entire system can be said to fail. Safety-critical real-time systems such as controllers found in avionic systems, as well as mission-critical real-time systems such as control systems on satellites fall within this class as they typically execute tasks with hard deadlines. Notwithstanding the emphasis on predictability over efficiency, these systems may be composed of tasks with tight timing constraints and low memory availability. Consequently, the worst case time for memory allocation, deallocation and access and the worst case space usage by the application must also be tightly bounded.

The predictability and efficiency of requests for memory allocation and deallocation is the focus of this thesis and solutions for efficient or predictable memory access will not be investigated.[1] It is therefore assumed that only one type of physical memory is used which exhibits uniform access speeds with no caching, paging or other similar technology being considered. This assumption is consistent with the predictability-oriented approach of real-time systems.

As already noted, modern practice dictates that hard real-time tasks should avoid allocating memory dynamically as guaranteeing the tight and bounded response times required of this class of applications is not easy. For example, the SPARK subset of Ada for high-integrity systems [17] subscribes to a strictly static memory policy. In this approach to memory management, the developer must know beforehand the maximum memory required by the application and preallocate that memory before any time-critical task begins execution. This approach is complex to implement and could incur higher memory requirements due to memory that is not in use, either because it has been preallocated but is not yet needed or has already been used but is no longer required. There are two solutions to this problem: the developer could either adopt a strategy of reusing memory or, alternatively, be able to dynamically allocate and free memory, thereby using the DMA during time-critical task execution. The former option is hard to achieve in all bar the simplest of applications. The typed nature of data also makes it hard for modern development practices to coexist with this approach. Dynamic memory allocation and deallocation is a suitable solution if the time for these operations is bounded and these bounds are tight enough to allow tasks to meet their timing requirements. Also, any space overheads of this approach must be small enough to compete with application-level reuse techniques, thereby reducing development complexity with minimal additional memory requirements.

The complexity of developing applications statically, as well as the high space overheads of this approach, necessitates a compromise: in return for a small but predictable increase in execution time overheads, a proposed dynamic memory model must decrease the combined development cost and space overheads of applications that target this model.[2] Unfortunately, the traditional approach to dynamic memory management – explicit allocation and deallocation at a fine granularity with an assumption of random arrival patterns – fails to achieve an acceptable tradeoff. Automatic memory management further exasperates this problem with additional overheads in both space and execution time. However, this thesis contends that a dynamic, coarse grain approach to memory management can help achieve this goal.

A coarse grain approach makes use of large blocks of memory that hold fine grain application data. These coarse grain entities called *regions* have longer lives than the required lifetime of the fine grain

[1] It is noted in passing that the response times on memory access are often hard to predict in the presence of technologies such as caches, though advances have been made in this regard with some processors providing features such as cache locking [120].

[2] The reason why development cost and space overheads are combined at this stage will be explained in Chapter 3.

data they contain but can also be allocated and deallocated dynamically. This provides immediate time benefits over allocation at a finer granularity as the number of allocation and deallocation requests is reduced to per-region allocation and deallocation. Although, this may result in an additional memory cost in comparison to a fine grain model as dead objects can only be freed once a whole region is freed, it can be shown that the reduction in fragmentation often results in lower overall memory requirements. Moreover, since information on the arrival patterns of allocation and deallocation requests are more easy to capture at a coarser granularity, it becomes feasible to adopt offline algorithms to order when or where memory is allocated according to the size and lifetime of memory requests. In this way, fragmentation can be further reduced, thereby resulting in even smaller space and time overheads. Compared to a static approach with application-level memory reuse, a coarse grain model can still provide tight space requirements and, crucially, a reduction in the development complexity of larger programs at the cost of a small and predictable execution time overhead. The development complexity in adopting a coarse grain model can be contained if the right tools for targeting this memory model are provided. Through an analysis of existing memory management solutions, the case for a coarse grain approach is made in Chapter 3.

1.2 Thesis Motivation, Goals, Hypothesis and Contributions

The motivation for this thesis comes from the recent debate about which memory model the RTSJ should adopt. Indeed, this continues to be the most contentious issue of the RTSJ. Java developers could adopt a static approach in the same way as the SPARK subset of Ada [17] but even the Ravenscar profile for Java [86] which also targets high-integrity systems recognises the need for a dynamic model by introducing a limited region-based memory model. The need for dynamic memory is however not a requirement only of real-time Java itself but is a requirement of the next generation of embedded real-time systems.

The first goal of this thesis is to evaluate the potential of the coarse grain approach in comparison to alternative memory models. A case will be made for why a coarse grain model can provide the required space-time-development cost tradeoff that static and fine grain explicit and automatic models fail to achieve. It is thereby argued that a coarse grain model could be attractive for applications of appreciable size and complexity with hard real-time requirements, tight and bounded space and time constraints and which require a comparably low development and maintenance cost. Proving the latter will require devising and evaluating a development approach to targeting a coarse grain memory model. The second goal of this thesis is to investigate how the space and time overheads of a coarse grain model are affected when the lifetime of regions is used to organise memory in a scoped (or stack) ordering as is done in the RTSJ. The third goal of this thesis is to apply the results of the previous two goals to the Java environment. The RTSJ adopts a scoped,

coarse grain memory model but this thesis will show that the specification requires a number of modifications in order to fully leverage the advantages of this approach. In particular, the emphasis will be on how this can be achieved in Java without the pitfalls of the RTSJ. In relation to these three goals, the hypothesis set to be proven in this thesis is divided into three parts and is stated as follows:

Part 1: As hard real-time systems become increasingly large and complex, forcing a static model of memory usage becomes untenable. The challenge is to provide a memory model that guarantees reasonably tight and bounded time and memory requirements without overburdening the developer with memory concerns. This thesis contends that a coarse grain approach can help *meet this challenge. In particular, program development and maintenance is aided by providing an environment that allows the expression of the lifetime aggregation of objects and the interaction of these aggregates as a separate concern to program logic.*

Part 2: Taking advantage of information about the allocation and deallocation pattern of regions is known to provide further savings in both the temporal and spatial dimensions. However, it can be proven that the strictly scoped model as provided in the RTSJ fails under certain conditions with memory requirements being infinite.

Part 3: Java applications can target the scoped, coarse grain model of the RTSJ without incurring the claimed high development complexity. However, this requires more functionality to express object lifetime, better use of this information by implementations and a development approach suitable for targeting this model. Nevertheless, given the potential failings of a strictly scoped model, alternatives must be provided in the RTSJ.

This thesis provides eight contributions in order to prove this hypothesis. These are:

1. A detailed investigation into the reasons why the fine grain approach to dynamic memory management introduces prohibitively large space and time overheads.

2. A novel approach to understanding the implications of using different memory models that automatically provides a classification of these models and attempts to settle the debate as to which model is best for different application classes. This is used to motivate a new approach to developing dynamic memory models that goes beyond the "fine-tuning" approach of current research. A coarse grain model in which the lifetime of regions are described relatively to each other is argued to be a positive step in this regard.

3. A model (called the *Region Partitioning Model*) for describing the lifetime of objects when

adopting a coarse grain approach to memory management. Memory usage in programs is specified using this model independently of the underlying functional logic. As required by the first part of the thesis hypothesis, this approach will prove key in reducing as much as possible the burden of memory management concerns the developer must contend with while leveraging the benefits of this approach.

4. An evaluation of a scoped approach to a coarse grain model and proof that the RTSJ model will eventually run out of memory for certain program behaviours described using the Region Partitioning Model. This is key in understanding why the model fails and motivating appropriate solutions.

5. An automatic scope ordering algorithm that minimises the memory requirements of complex programs; this contribution is a by-product of the previous contribution.

6. Three extensions to the RTSJ – two to the RTSJ model and Application Programming Interface (API) and one to the specification requirements of implementations – that are motivated by the previous contributions and described within the context of the Region Partitioning Model.

7. An overview of the implementation of:

   (a) the Region Partitioning Model on top of the RTSJ scoped memory model,

   (b) the proposed extension to the implementation requirements of the RTSJ memory model and

   (c) the two proposed extensions to the RTSJ API.

8. An example application that illustrates how a developer would make use of the Region Partitioning Model to develop applications that target a scoped, coarse grain model in the RTSJ.

1.3 Proving the Hypothesis: The Thesis Plan

This thesis is set out as follows: Chapter 2 provides an in-depth investigation of current approaches in real-time memory management and a review of related literature on the RTSJ memory model. Although the results described here make a strong case for an alternative memory model to traditional fine grain approaches, current research either fails to provide models for which the overheads are sufficiently tight or proposes models which place a significant burden on the developer. A brief description of the intuition behind the adoption of a coarse grain approach is begun in the discussions in Chapter 2 and continued in Chapter 3 where a set of criteria for evaluating a memory model is introduced and a unified approach to distinguishing the merits of each memory model is developed. This approach is based on the *Entropy Hypothesis* that allows the derivation of a classification of

these models based on a hypothesis that highlights the relationship across three semantic domains: space, time and a characterisation of available information about the way memory is used by the application. The Entropy Hypothesis is used to make a case for the direction research needs to take in order to develop memory models suitable for the next generation of complex real-time and embedded systems. In particular, it is argued that rather than trying to adapt the memory models used in traditional environments to real-time ones, new memory models that directly address real-time requirements need to be developed. The conclusion is that applications that target a coarse grain model should be able to achieve the tight space and time constraints of hard real-time systems. The caveat here is that choosing to target this model for a particular application does not over-burden the developer with memory concerns.

In order to attain the benefits of the coarse grain approach as argued through the Entropy Hypothesis, the Region Partitioning Model for expressing region lifetime is developed in Chapter 4. The Region Partitioning Model provides a mechanism for defining region lifetime as a cross-cutting concern, thereby separating memory concerns from application code. This is an important step in providing a mechanism that aids the development and maintenance process when targeting coarse grain models, thereby addressing the first part of the thesis hypothesis. By applying different coarse grain implementations to the Region Partitioning Model, this abstraction allows an evaluation of the space and time requirements of each model. The application of the Region Partitioning Model to an unscoped, coarse grain memory model called RPM − Unscoped is carried out in this chapter.

Chapter 5 uses the Region Partitioning Model to investigate the effect of a scoped ordering of regions. A memory model called RPM − Scoped is introduced that places constraints on the allocation and deallocation points of regions in order to achieve this scoped order. The advantages and pitfalls of RPM − Scoped are deduced and the second part of the thesis hypothesis is proven. This requires an offline algorithm that automatically derives the minimum memory requirements of an application targeting RPM − Scoped by finding an ideal scope ordering. A comparison of RPM − Scoped to the RTSJ scoped memory model is provided and is used to motivate two extensions to the RTSJ, one in the semantics of the model at the API-level and one in the requirements specification of implementations.

The next two chapters of the thesis form the basis of the proof of the final part of the thesis hypothesis. Chapter 6 makes concrete the theoretical Region Partitioning Model by developing classes that allow developers to target the Region Partitioning Model in an RTSJ environment. A second API-level extension to the RTSJ's memory model is proposed that retains Java's reference safety but without the restrictions imposed by the scoped reference rules. In Chapter 7, the thesis is evaluated within the context of the thesis hypothesis. An example application is developed to show how a real-world application would be developed using the extended RTSJ and Region Partitioning

Model classes. A criticism of the scoped approach to coarse grain memory management is also provided. Finally, Chapter 8 concludes with a summary of the contributions in this thesis and related areas of research that warrant further investigation.

Chapter 2

Memory Management in Real-Time Applications

The Java programming model has become established in mainstream software development as a platform for general-purpose applications. The abstraction provided by the virtual machine realises the attractive Write-Once-Run-Anywhere (WORA) concept. Initially created for developing software for consumer devices, it achieved its success in the rapid expansion of the Internet for which the WORA concept found an ideal application. Today, Java is penetrating into more niche markets, from large enterprise applications to small embedded devices such as mobile phones. Until recently however, "Java" and "real-time" were considered an oxymoron as Java lacked the infrastructure to enable development of real-time systems. The specifications of both Language [69] and Virtual Machine [89] were never designed with timeliness as a key issue. The language therefore fails to allow more advanced temporal requirements of threads to be expressed and virtual machine implementations may behave unpredictably in this context. For example, whereas a basic thread priority can be specified, it is not required to be observed by a virtual machine and there is no guarantee that the highest priority thread will preempt lower priority threads. In order to address this shortcoming, two competing specifications have been introduced: the "Real-Time Core Extensions" from the J-Consortium [115] and the "Real-Time Specification for Java" (RTSJ) [68] from Sun Microsystems. The former has not achieved the exposure that the RTSJ has, though attempts are now being made to combine the two standards for a safety-critical Java [46].

The motivation for a Java development platform for real-time systems may be questionable. Indeed, there already exist several well established platforms for developing real-time systems. Many of the extensions to the RTSJ such as priority schedulers and synchronisation protocols have already been implemented in a number of operating systems and runtimes with appropriate APIs exported for developers. The C programming language [81] and its object-oriented counterpart, C++, are widely used as a programming language in the real-time domain with developers making use of

real-time libraries, operating systems and middleware. Today, C and C++ are increasingly being used solely for specialised systems programming and, critically, industry players often report a growing shortage of developers. Unlike Ada and Java, there is no well defined runtime paired with C and C++[1] and this results in high platform dependency and high development and maintenance costs. The Ada [113] programming language and the requirements specified by its runtime were specifically intended to meet the needs of embedded and real-time applications. For general purpose programming, Ada has failed to progress sufficiently to challenge more modern languages such as Java and C# [4],[2] partly due to a lack of corporate investment. The next version of Ada [93] sees significant changes to the language and its semantics that should put it on par with Java. Of significance is that several of these extensions adopt Java-like approaches, interfaces being an example of this. Even in the area of real-time programming in which it has dominated for over a decade, Ada is threatened by the RTSJ as the language of choice for real-time developers. In [122], Wellings argues that the RTSJ provides language-integrated real-time mechanisms that make it superior to the low-level, language-orthogonal approaches adopted in the current version of Ada. The new version of Ada will address these issues in a bid to retain its dominant position.[3]

As a relatively young technology in a very specialised field, the RTSJ is still some distance away from being adopted in mainstream real-time development. A number of minor ambiguities in the specification and an absence of stable and efficient implementations[4] have restricted uptake. While this has detracted from the RTSJ's appeal [5], these are problems that can be addressed in the short term. The attraction of Java to the real-time arena comes from its traditional strengths, namely: its portability, memory model (and resulting pointer (reference) safety), object-oriented paradigm, strict typing and in-built threading model. In order to succeed in the longer term and challenge the *status quo* of real-time systems development, the RTSJ must provide equal functionality in comparison with the competition while maintaining these strengths. In some respects, the RTSJ has achieved this goal and delivers an API that provides a broad functionality, incorporating real-time features that are well understood in other languages and operating systems. Java's object-oriented model is not modified and the threading mechanism is not changed radically but extended to allow real-time requirements to be expressed. Although WORA is maintained to the extent that physical memory access is avoided, acknowledging that this is required in real-time systems by providing Java classes

[1] This is what standards such as real-time POSIX [76] attempt to provide.

[2] The runtime for C# is the Common Language Runtime (CLR) within the .NET framework [121].

[3] As with the base language, several mechanisms that will be introduced are similar to those that already exist in the RTSJ, asynchronous event handling being an example. This is expected as advances in one language are recycled and improved in more recent versions of the other.

[4] There are a number of implementations of the RTSJ but none with the integrity required of real-world environments, particulary high-integrity and safety-critical environments. Most implementations are developed as research projects within academia, examples of which include FLEX [22], OVM [98] and JRate[54]. Earlier versions of the reference implementation [1] were unstable and did not implement several key methods. The first industry-standard implementation is being developed by Sun Microsystems [6] and should allow real-time developers to realistically consider Java.

to access physical memory is a positive tradeoff. The one area in which critics of the RTSJ claim that it fails to deliver is in the area of safety as a result of significant changes to its memory model. Indeed, the memory model continues to be the most contentious issue of the RTSJ and more literature exists on analysis, extensions and modifications of the RTSJ memory model than any other part of the specification.

Java is described as a language that provides "ease-of-use" in application development. This has been an important component of the language's success and is attributed in particular to the safety it provides in its type system[5] and the elimination of memory management concerns from the user domain. However, the RTSJ makes significant changes to its memory model that have resulted in Java's reference safety being compromised and the user burdened with memory concerns. Moreover, memory-intensive applications are sometimes more complex to develop than they might have been had an explicit, fine grain memory managed model been introduced instead. Indeed, critics of the RTSJ have argued that the scoped memory model is unworkable and is one of the main reason why it has failed to attract real-time and embedded systems developers [5].

This chapter explores the memory management problem in real-time systems and how the RTSJ memory model addresses this problem. Existing work on this memory model that address the problems it itself introduces are then discussed in turn. Section 2.1 investigates the rationale behind the changes to the Java memory model. Clearly, such sweeping modifications to the current model through the introduction of a novel alternative so distinct from traditional fine grain approaches must be necessitated by real-time requirements. The memory management problem in real-time environments is discussed and existing fine grain solutions are described in detail. Initial conclusions indicate that fine grain memory management solutions fail to deliver adequately tight space and time overheads for this application domain and further research into these models will in all probability provide only marginal improvements. This theory is expanded upon in the Chapter 3. Section 2.2 provides a brief introduction to the RTSJ with particular emphasis to the parts of the API relevant to memory management. Section 2.3, describes existing work that has attempted to simplify the RTSJ memory model and the way it is used by developers. An overview of existing implementations of the model as well as research into region-based memory management outside of the RTSJ is also provided in this section.

[5]Strict typing allows many errors to be found at compile time while Java's memory model guarantees that dangling references are never created.

2.1 Porting Traditional Fine Grain Dynamic Memory Management to Real-Time Systems

The problem of memory management in hard real-time systems has already been introduced briefly in Chapter 1. The traditional way of developing hard real-time systems was said to be that of avoiding dynamic memory allocation in critical code altogether. This is the approach taken in high-integrity and safety-critical subsets of Ada such as SPARK [17].[6] Assuming an object-oriented language, the application would allocate all necessary objects in a pre-mission phase and these would live for the duration of the mission phase. The limitations of this approach become apparent in applications of significant size and complexity. For example, if an application needs n_1 objects at a particular time of type t_1 and n_2 objects of type t_2 at another, then $n_1 \times sizeof(t_1) + n_2 \times sizeof(t_2)$ bytes are required using the static approach. If dynamic allocation and deallocation is allowed however, as little as $\max(n_1 \times sizeof(t_1), n_2 \times sizeof(t_2))$ bytes may be required. If memory constrained devices are the target of the application, developers are often forced to consider recycling objects, thereby adding another dimension of complexity to the development and maintenance processes. In some cases, this problem is aggravated as the object model may need to be broken in order to allow what would ordinarily be type-incompatible objects to replace each other. Dynamic memory allocation and deallocation can address this problem if the reduced development complexity does not come at an unacceptable space and time overhead.

Dynamic memory management is used in most development environments outside the real-time domain in the form of user-controlled allocation and deallocation and automatic deallocation by a garbage collector. The allocation and deallocation strategies in these models is generally carried out by an "online" Dynamic Memory Allocator (DMA) which does not take advantage of any available explicit information about the arrival pattern of allocation and deallocation requests. It is unsurprising that recent research focuses for the most part on porting these models to the real-time domain. However, in proposing the use of these memory models in this domain, the space and time overheads need to be quantified. This is investigated in the rest of this section and subsequently used to motivate research into an alternative memory model. Note that an exact evaluation of each overhead is often not possible due to the complex dependencies of the space-time tradeoff. This problem will be addressed in detail at the end of this section. The overall goal here is to give some insight into the impact each overhead has on an application and as well as their relative cost:

1. Both explicit allocation and deallocation and automatic memory models introduce two overheads:

   (a) memory fragmentation, a phenomenon caused by the allocation and deallocation policy of

[6] An example of adopting SPARK Ada in a real-world scenario is given in [45].

the DMA that introduces unpredictability and high pessimism in the space requirements of applications and significantly increases the response time of allocation requests and

(b) the time overhead of pushing allocation and deallocation to the critical execution phase.

2. In environments that perform automatic rather than explicit deallocation there is the additional overhead of identifying garbage.

3. If memory management is made incrementally, an additional overhead for guaranteeing the mutual integrity of the program and collector is incurred due to concurrent manipulation of the object graph. While this overhead could be applied to both explicit and automatic memory models, it is normally only found in the latter as explicit memory management is rarely implemented in this way.

4. In order to counter (1a), both memory models can implement a defragmentation routine. This results in additional temporal costs but reduced space requirements. Again, memory defragmentation could be applied to both explicit and automatic memory models but is rarely found in the former.

2.1.1 The Cost of Memory Fragmentation

In order to describe the fragmentation problem, this section first considers a minimal system model which consists of a single application that is loaded completely into memory before execution begins. The program has a memory area available to it that is of a predetermined size and is fixed throughout the lifetime of the application. The system utilises an online algorithm called the Dynamic Memory Allocator (DMA) that allocates and deallocates memory from this memory area as requested by the application and for which the arrival patterns (size, frequency and order) of these requests is unknown in general or can not be expressed in the memory model's API. The DMA therefore exposes an API that provides just two functions, one to allocate memory and one to free memory. The allocation function reserves a block of memory for use by the calling application. This function takes a single integer parameter that specifies the number of bytes to allocate and returns a handle back to the application in order for it to use the memory space. For example, the POSIX standard provides the `void*` *malloc(*`int size`*)* system call that allocates `size` bytes and returns a pointer to the allocated memory or `null` on failure. For object systems such as Java, there is an extra level of abstraction so that when a *new ObjectType()* invocation is made by the application, an equivalent *malloc(sizeof(*`ObjectType`*))* call is made in the runtime and a virtual reference is returned to the application rather than a direct pointer into memory. While this approach eliminates the possibility of using pointer arithmetic, it allows the DMA to move memory at runtime should this improve performance or make more efficient use of memory. When servicing an allocation request, the DMA

is responsible for selecting a section of memory from the total memory allocated to the application to service the call. If the DMA identifies any memory that is no longer required by the application, then that area could be reused. This would be the case either because the application issues a deallocation call (such as the `void` *free(*`void *ptr`*)* system call in POSIX) or through some runtime analysis that establishes that the application no longer needs the memory in question (that is, through garbage collection).

Memory fragmentation occurs due to allocation requests of different sizes and subsequent deallocation in arbitrary order. Consider an application which has 11 units of memory available to it. When the application makes allocation requests of varying sizes, the DMA reserves memory according to some given policy. One approach is to use a "first-fit" algorithm that allocates the first available block of memory that is large enough to fulfill the request.[7] Figure 2.1 shows a sequence of allocation and deallocation requests and their execution by a DMA using the first-fit policy. Fragmentation occurs when the available free memory is not in one contiguous block. This occurs in Steps 6, 7 and 8 in Figure 2.1 where 2, 4 and 3 blocks of memory are available respectively. Due to memory fragmentation, allocation requests can only be satisfied for requests of size 2 after Step 7 and of unit size after Steps 6 and 8. The fragmentation shown in this example is called *external fragmentation*. Most DMA algorithms divide memory into blocks which can not be shared across requests. Therefore, if a block size of n bytes is used to service an allocation request for m bytes ($m \leq n$), then $n - (m \bmod n)$ bytes will be wasted. This phenomenon is called *internal fragmentation*. In order to address both internal and external fragmentation, DMAs implement a number of strategies such as block splitting and coalescing. Buddy algorithms [83] are an example of a block allocator which attempts to minimise both internal and external fragmentation by dividing and coalescing adjacent blocks as necessary.

Memory fragmentation introduces high pessimism and unpredictability in space and time requirements that is unfavourable in real-time environments [88]. If available memory becomes highly fragmented, the time required to find an available block of contiguous memory suitable for a request may be too long, thereby proving unsuitable for real-time environments with tight timing constraints. High fragmentation also leads to memory wastage that can in turn lead to an application running out of memory. In real-time environments, the worst case memory requirements must also be calculated and not be prohibitively high. An investigation into the results from past work is carried out next in order to quantify these overheads.

[7] There exist several DMA algorithms that aim to optimise for a particular pattern of memory usage. As a description of these algorithms is beyond the scope of this thesis, the reader is referred to [114] where an adequate summary can be found.

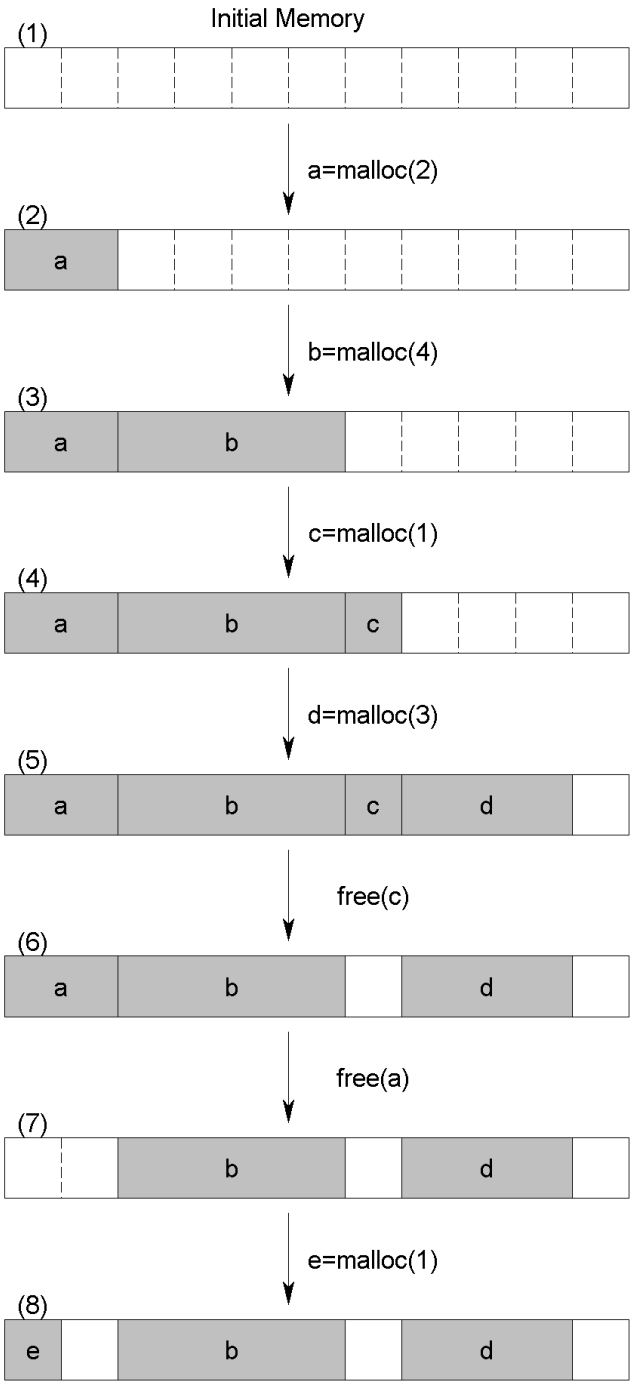


Figure 2.1: External Memory Fragmentation in a First-Fit DMA

Measuring the Cost of Fragmentation: Space

An important prerequisite of this discussion is to define fragmentation. It is in fact of little use to define fragmentation as a function of the number of "holes" in the memory space. If this space were to consist of alternating blocks of full and empty cells of unit size, then fragmentation might be deemed to be high. However, if at this point, all subsequent requests made are of unit size, then fragmentation is not a problem. It is only when a request twice unit size is made that fragmentation results in further memory being needed. Rather than describing fragmentation in terms of characteristics of the pattern of memory usage, fragmentation is therefore defined in ways that give an indication of the amount of memory required during the application's lifetime. A useful proposition by Neely [95] and Johnstone [77, 78] is to separate the overheads caused by the book-keeping data structures of the DMA from the extra space required to guarantee that allocation requests are satisfied in the presence of fragmentation. This separation allows consideration of the effects of *policy* and *mechanism* to be analysed independently. For example, the best-fit algorithm is a DMA policy that can be implemented by a number of mechanisms such as sequential searching of a free list data structure or indexed searching of a binary tree representation of the free memory. A DMA policy incorporates a number of *strategies* that aim to reduce fragmentation through what Johnstone describes as "regularities in the request stream". For example, the decision when to split and coalesce blocks and the choice of block sizes in the segregated free list can both reduce fragmentation and are used to supplement the basic policy.

Neely [95] described the space overhead as a *relative fragmentation* that is calculated as a percentage of the maximum number of live objects (that is $F = \frac{MaxHeapSize - MaxLiveBytes}{MaxLiveBytes}$). Using this approach, the maximum fragmentation does not necessarily occur when *MaxLiveBytes* are required. A similar approach used elsewhere in the literature [67, 82, 94] is to describe the order of magnitude increase in requirements compared to an optimal solution. This value is called the *competitive ratio* [40] for which the optimal solution in a DMA can be assumed to be that in which *MaxHeapSize* is as close as possible to *MaxLiveBytes* and therefore fragmentation is as close to zero as possible.[8] Neely's work forms the basis for a more detailed analysis by Johnstone [77] in which other aspects of DMA allocation strategies such as locality of reference are investigated. Johnstone uses a different metric, comparing the memory requirements only at the point of *MaxLiveBytes* (that is $F = \frac{MaxHeapSizeAtMaxLiveBytes - MaxLiveBytes}{MaxHeapSizeAtMaxLiveBytes}$). Neely's approach and the competitive ratio are perhaps more relevant as they consider the actual space required throughout the lifetime of the program. However, it can be assumed that *MaxHeapSize* does in fact occur at the same time as when *MaxLiveBytes* occurs, an observation Neely claims to be close enough in his work and which

[8] For the purposes of this discussion it can be assumed that Neely's fragmentation factor F is the same as the competitive ratio so that the optimal allocation strategy results in zero fragmentation. In practice, it is known that an optimal offline algorithm can achieve at best a competitive ratio of 5/4 [15, 82].

is further corroborated by Johnstone's results.

The results from Johnstone's work are intriguing. In this and other work [78], Johnstone's results show that fragmentation due to policy is least in the simpler policies such as first and best fit as opposed to more complex policies such as buddy algorithms. These simpler algorithms exhibit fragmentation that is also very low, typically under 3% when averaged across all applications. In the binary buddy algorithm, fragmentation is as high as 53%, probably due to the greater possibility of internal fragmentation and decreased possibility of coalescing.[9] When taking into consideration the implementation costs of the policy (the mechanism) and machine requirements (namely byte alignment), these overheads increase to about 34% for best-fit and first-fit policies and 74% for binary buddy. Around equal amounts of this overhead is attributed to mechanism and machine requirements in the case of best-fit and first-fit DMAs.

A conclusion that Johnstone draws from his work is that the fragmentation problem is solved, and has been solved for several decades. This is because the overheads of DMAs comes not from the policy itself but from the inevitable overheads of the implementation's book-keeping costs and the machine particularities. The failure of researches to realise this is attributed to using synthetic traces [66] rather than real-world traces, and the failure to separate policy from mechanism. This latter problem in particular resulted in research being carried out in finding new policies, rather than better implementations of the already existing good policies. Indeed, the fragmentation due to policy is nearly zero in known algorithms and implementations of these mechanisms result in space overheads that are still significantly lower than the fragmentation due to policy alone of algorithms such as binary buddy. Research should therefore be directed at better implementations of policies that exhibit low fragmentation. An important observation Johnstone makes is that understanding the strategies used in the better performing policies could help develop even better all round DMAs with lower cost mechanisms.

Johnstone's work is based on the observed rather than worst case space requirements of applications.[10] As already discussed, real-time applications need to consider the analytical worst case requirements in order to provide the necessary guarantees. While it is true that the worst case is seldom observed, real-time applications are typically non-terminating and the occurrence of the worst case is a significant possibility. The worst case space requirements of DMA policies have been analysed in earlier work in the 1970's and 1980's but there has been little work on this area since then. The first related work is probably Graham's[11] who described the fragmentation problem in terms of a game between an attacker A and a defender D. A moves in two phases: in the first he removes strings of squares from the board and in the second he forms new strings which D must then

[9] This is further impressed by the even poorer performance of some segregated policies.

[10] The applications tested range from the widely used Ghostscript postscript interpreter and Espresso optimiser for programmable logic arrays to a mathematical application called Grobner.

[11] Unfortunately, this work is unpublished but is described by Wilson *et. al.* in [125].

place on the board. The board size is analogous to the size of the heap and a limit on the maximum size of the new strings is also given. Fragmentation is a function of the size of the total number of busy words (denoted as M here and as *MaxLiveBytes* in Johnstone's work) and the maximum block size (denoted as n). This differs from Johnstone's work in that block size is also considered but this value could easily be plugged in from a program analysis.

In a series of work between 1971 and 1977 [103, 104, 105], Robson derived the worst case memory requirements of the best-fit and first-fit policies whereas Knowlton [83] derived in 1965 the worst case for buddy systems. Given a maximum block size n and maximum live memory requirements of M, the worst case memory requirements are $M \log_2 n$ for first-fit, $2M \log_2 n$ for binary buddy and Mn for best-fit.[12] Luby *et. al.* [90] showed that the worst case requirements for some segregated policies is similar to that for first-fit. Crucially, Robson showed that there exists an optimal strategy for which the worst case memory requirements lie between $\frac{1}{2}M \log_2 n$ and about $0.84M \log_2 n$. The first-fit policy therefore provides a solution that is very close to optimal. Note that Robson's, Knowlton's and Luby's work only account for the overhead of fragmentation due to policy. In particular, the space costs of the implementation add a further space overhead that is proportional to the maximum number of possible free blocks.

An interesting exercise is to compare Robson's value for the lower bound in the worst case to Johnstone's results for observed fragmentation. Though care must be taken as the metrics used differ slightly in these two studies, realistic assumptions such as that the request for the largest heap size occurs at *MaxLiveBytes* allows for a direct comparison to be made. Clearly, both Johnstone and Robson lead to the conclusion that the first-fit policy is the best solution both in terms of observed and analytical worst case overheads. The results for best-fit are in stark contrast however, with the observed worst case being far better than the analytical worst case. This is an example in which the possibility of the worst case occurring is so rare that it does not show up in real-world applications. Binary buddy systems perform badly in the observed worst case and are mediocre in the analytical worst case. However, the most significant observation that can be drawn by comparing Johnstone's and Robson's work is that there exists a large discrepancy between the observed and analytical worst cases for all policies.

It would be tempting to conclude that a fine grain memory model using a first-fit policy would provide the best solution for real-time applications. In fact, the worst case memory requirements of a first-fit policy are still rather high. For example, a program with a maximum live memory requirements of 1Mb and which allocates objects that range over a conservative size (say between 1 byte and 1K bytes) would still require 10Mb to guarantee against breakdown due to fragmentation.

[12]The assumption here is that the minimum block size required is 1. In reality, both the smallest and largest block size need to be considered. For example, for the first-fit policy, the worst case memory requirements are given by the binary logarithm of the ratio of the largest object size divided by the smallest object size.

In addition to this, the memory requirements of the mechanism implementing the policy must also be considered. Even assuming the observed overheads are approximately equal to the analytical worst case overheads, this would add roughly a further 50% to the memory requirements according to Johnstone. These total memory requirements are a significant order of magnitude higher that the 1.5Mb to 2Mb requirements one would expect to be required to cover both policy and mechanism in the observed case. It is important to note that Robson's derivation of the worst case applies to programs executing for a sufficiently long period of time to reach this worst case. Johnstone's is based on real-world terminating applications that are not typical of the real-time domain.

Measuring the Cost of Fragmentation: Time

The worst case time overheads of different DMAs is measured in relation to the mechanism used to implement the chosen policy. Unlike the analysis for space overheads, policy and mechanism can not be separated when considering time overheads. There has been very little research carried out in this area in the past, probably as a result of the apprehension of real-time developers in using dynamic memory models in the first place. In [97], Nilsen *et. al.* measure the observed worst-case time overheads of a number of DMAs used in C and C++ environments. The results of this work are rather hard to compare with the related DMA policies as the effects of initialisation costs of the DMA and the use of caches are not eliminated from the analysis. Also, the DMAs analysed use a combination of policies and strategies that make it hard to give a clear indication of where the sources of overheads lie.

The most important contribution in the analysis of the time overheads incurred by DMAs was provided recently by Puaut in [100]. Here, the average and worst observed times of four applications using a number of different DMAs are compared to the analytical worst case allocation and deallocation times of these algorithms. The results of the observed mean and worst performance confirm a theory of Johnstone's that the better performing policies such as best-fit and first-fit can be implemented efficiently. Indeed, the average observed time overheads are similar across all DMAs and moreover, the worst case overheads are typically less in the simpler policies such as best-fit than in the more complex ones such as binary buddy and Fibonacci buddy [74]. The analytical worst case overheads however tell a different story. Here, the analytical worst case performance of best-fit and first-fit DMAs using a naïve mechanism is nearly a thousand times worse than that of the buddy systems. This is because in the rarely occurring worst case in best-fit and worst-fit systems, the entire heap needs to be traversed in order to find a suitable space for allocation; in buddy systems this is never the case. Johnstone's emphasis on the importance of a well-designed implementation of a policy is highlighted here with a seventy-fold improvement in the analytical worst case of the best-fit policy by utilising a binary-tree to maintain a record of free locations. Although this is achieved with only a minimal degradation in the observed worst case allocation times, it is still 13

times worse than the analytical worst case of buddy systems. The percentage time overhead compared to the static approach would depend on the inter-arrival rate of allocation and deallocation requests and the cost of an allocation and deallocation cycle as well as the mean and variance of the request size. Based on the results in [100],[13] the significant proportion of time used by the DMA in a typical program is immediately evident: the values for b-tree best fit and binary buddy would be respectively around 64% and 63% time overhead for a logic optimisation program called Espr; that is about 63% and 64% of program execution is used in servicing allocation and deallocation requests. In the worst case, these values jump to 96% and 71% respectively.

The result of Robson's, Johnstone's and Puaut's work allows for a comparison to be made between the different allocation policies and the overheads they incur, both in the observed and analytical worst case. This provides important insight into the selection of an appropriate DMA for real-time applications with particular space and time requirements. However, as will be discussed in Section 2.1.7, the absence of a theory consolidating this work makes it hard to motivate new DMA solutions.

2.1.2 The Cost of Allocation/Deallocation Calls

In a static approach to memory management, allocation is carried out at a pre-mission phase. Although this incurs high startup costs, no extra space is required at runtime and, crucially, time overheads are zero during the time-critical mission phase. Any memory model that allows memory to be allocated and freed at runtime incurs a per-allocation/deallocation time cost irrespective of fragmentation or any other overheads. For allocation, this is the cost of the vertical switch into the runtime, updating the free list or relevant data structure once a location has been found, initialising the allocated space and finally vertically switching back to the application. Although, this cost is normally absorbed into the time overheads of the DMA mechanism when comparing memory models, separating this overhead as a separate concern identifies the tradeoff between the high startup costs of the static approach and a contributory factor of the additional runtime costs of using a DMA. To evaluate the allocation part of this cost, consider that if the memory available were infinite, then the time overhead would be equal to the number of cycles required for allocation alone as deallocation could be ignored. The function of the DMA is therefore reduced to a single addition and pointer update together with some initialisation at allocation. If deallocation requires a similar overhead, then the combined value is still small in comparison to the full cost of a DMA. Nevertheless, these minimal runtime costs in a dynamic environment accumulate to significant startup costs in the static approach.

[13] There is an assumption here that the implemented mechanisms are of a sufficient standard and properly optimised. There is no reason to believe that this is not the case to a degree that would jeopardise the integrity of the results.

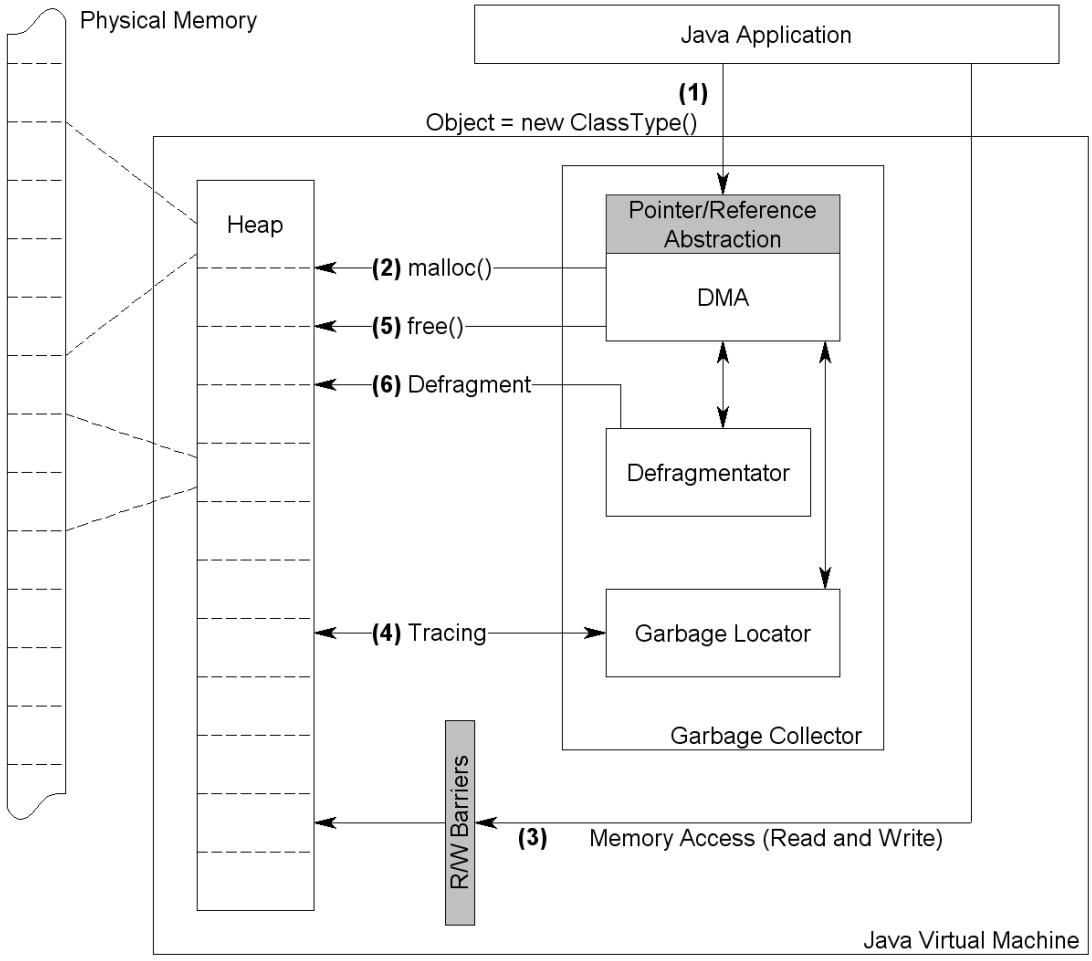
2.1.3 The Cost of (Non-Incremental) Automatic Memory Management

Automatic memory management (or garbage collection) is used in memory models in which the DMA's API exposes an allocation function but no deallocation one. Automatic memory management algorithms (garbage collectors) are therefore required to supplement the DMA to identify memory that is no longer being used by an application, mark this memory for deallocation and subsequently invoke the DMA to free this memory. It is often implied that the garbage collector also assumes the role of the DMA and defragmentor, thereby executing four tasks: servicing allocations, locating garbage through a root scan and traversal of the object graph (*tracing*), freeing garbage (*sweeping*) and defragmentation. This blurs the distinction between DMAs and garbage collectors and limits a direct comparison between explicit allocation and deallocation memory models and automatic memory management models. In an effort to quantify each overhead, this thesis maintains the distinction between the processes of allocation, tracing, sweeping and defragmenting memory. For simple mark-sweep garbage collectors [29], this separation is clearly defined and the collector can be decomposed into three components:

1. a tracing algorithm that identifies garbage,
2. a DMA that makes *malloc()* requests when *new()* is invoked to allocate memory for a new object and *free()* calls when an object is marked as unreachable during a garbage and collection cycle
3. the defragmentation routine that copies memory in order to increase the size of free contiguous blocks.

Figure 2.2 depicts this general arrangement within the context of a Java environment. For some collectors, these processes are harder to separate and therefore to quantify independently. For example, in semi-space collectors [16] tracing is carried out in a similar fashion to mark-sweep collectors and the DMA makes fine grain allocations in to-space. Conceptually, defragmentation is carried out before deallocation by first copying live data between spaces and then executing a coarse grain deallocation by switching the spaces.

There is an additional overhead in a garbage collected environment over an explicitly managed one serviced directly by a DMA that is highlighted by this abstraction: there is a time overhead involved in tracing that is not present when memory is managed explicitly and a space overhead due to the delayed deallocation of memory. Given this additional overhead, it would be expected that garbage collection would automatically imply higher total overheads than an explicit memory management, in both the space and time dimensions. There are several cases in the literature in which this is argued not to be the case in the time dimension. In [28], Boehm argues that the ideal memory model to use in some application depends on a number of factors including the size of objects in that application and the available heap space. In [72], Hertz *et. al.* show that



(1) Application instantiates a new object

(2) Memory for object is allocated by the DMA

(3) Integrity is maintained using read/write barriers

(4) Tracing identifies unreachable objects

(5) Unreachable objects are deallocated by the DMA

(6) Defragmentation occurs if a fragmentation threshold is reached

Figure 2.2: Separating Concerns in Garbage Collection Environments

garbage collectors can achieve and even surpass the efficiency of explicit memory management but confirm that garbage collection requires six times the physical memory to achieve this performance benefit. This phenomenon occurs because the delayed deallocation of objects in a garbage collected environment results in higher *space* overheads but incurs lower time overheads due to infrequent vertical switching between the application and underlying memory subsystem. These temporal savings can be large enough to offset the costs of tracing. However, existing garbage collectors make use of strategies that are absent in existing DMAs but that could be readily implemented. The arguments made by Boehm and Hertz fail to capture this.[14] For example, it could be argued that a similar technique to reduce vertical switching could be used for explicit memory management with *free()* calls being delayed and a single call to the DMA passing the addresses of all memory to be freed. Hence, the performance of an explicitly managed environment can always be made to be at least as good as that of any garbage collector by using the same strategies to complement the DMA. Garbage collectors will however always incur the additional overheads of tracing. This is an important observation as the cost of tracing becomes the single additional overhead between explicitly managed memory models and non-incremental automatic memory management models. Since the singular goal of using a tracing algorithm is to guarantee reference safety, these overheads indicate the cost of maintaining reference safety.

The cost of tracing depends on the frequency of invocation of the collector and the amount of live memory. The time overhead is proportional to the amount of live data and not the total amount of memory in use, including garbage. The space overheads depends on the frequency of invocation of the collector in relation to the allocation rate. At one extreme, the collector is never called and therefore the cost of tracing is zero and the space overheard is equal to the amount of garbage generated. At the other extreme, the garbage collector is invoked frequently enough to require a trace proportional to the amount of live data after every allocation and pointer dereference and additional space overheads are zero. The results in [12] show that for a program with live memory of around 80Mb, the time for a trace is about 2 to 3 seconds, the majority of which is taken up by the mark phase rather than the root scan of the tracing process. It is emphasised that these costs are in addition to the space and time costs of the DMA and the time cost of sweeping.

2.1.4 The Cost of Incremental Collection

Although theoretically any garbage collector can be used in a real-time environment if the worst case pause time is known, this is academic if these pause times are too large. Stop-the-world collectors such as the simple mark-sweep-defragment collector could experience pause times that run into seconds. Real-time collectors are therefore necessarily incremental, interleaving their execution with

[14] In fairness, the goal of this work was to compare existing DMAs and traditional non real-time collectors rather than to understand or address the source of overheads.

that of the application (or mutator) in order to spread out these pause times across the application's execution. This is required as the point at which garbage collection becomes necessary is usually unknown and, therefore, the cost of a garbage collection cycle must be applied to all possible locations at which collection might be triggered. Even if this point is limited to allocation sites, the worst case execution time of tasks becomes prohibitively pessimistic if pause times are too large. Before investigating the cost of an incremental approach over the stop-the-world model and comparing the two, a brief introduction to real-time garbage collection is necessary.

Understanding Real-Time Garbage Collection

The definition of what constitutes real-time garbage collection has proven contentious. These definitions qualify a garbage collector based on how pause times are spread out across program execution. Baker's seminal paper [16] provides the first definition of real-time garbage collection as one in which the time required for memory operations is bounded by a *small constant*. This approach is simple in that the cost of this constant is added to every memory operation making it possible to derive the worst case execution time for schedulability analysis. The disadvantage of this approach is the possibly severe drop in mutator time (that is the time when the program actually makes progress) when memory operations occur close to each other. One solution to this problem is to redefine which memory operations result in collector execution. Brooks' variant of Baker's algorithm [42] in which the costly barriers that result in data copying are only executed on less frequent writes is an example of this.

The second definition of real-time collection requires not only that pause times are low but that the minimum mutator utilisation (MMU) is bounded from below and is consistent throughout execution lifetime [12, 77]. This means that the pause times must not only be small but must be spread out sufficiently to allow the application to progress. Therefore, at the cost of reduced mutator utilisation at points of infrequent memory operations, memory intensive parts of the program are not subject to too many expensive pauses. Advocates of this second definition describe the former definition as flawed. This is not correct in the context of the definition of a real-time system as it may well be the case that the cost of memory operations is accommodated for in the worst case execution time analysis and sufficient for the timing requirements of the system. Admittedly however, the typical 10 to 20 times overhead of a normal read operation in comparison to the execution cost of a read barrier [12] does make a dispersal of these costs across the application irrespective of the cohesion of memory operations attractive. If a single task makes a system unschedulable due to high time overheads of memory operations, offloading some of this overhead to other points in the program may make the system schedulable. In light of this, two different approaches to incremental garbage collection have been developed: time-based scheduling and work-based scheduling. The former approach interleaves the process of identifying and freeing garbage and defragmentation with

application execution on a schedule whereas the latter approach carries out a little work at specified points in the program execution (typically at read and write) in the spirit of Baker and Brooks.

Time-based scheduling is based on a global characterisation of the properties of the system such as the memory allocation rate, the rate of collection and the size of available memory. Using these values, a required execution time for the collector can be calculated within a traditional schedulability framework. Henriksson's thesis [102] describes such an approach, calculating the worst case response time of the garbage collector as a medium priority thread using Joseph and Pandya's analysis [79] after having calculated the worst case execution time of the collector. Herikkson opts to offload garbage collection interruption to lower priority threads, thereby allowing higher priority threads to avoid garbage collection penalties. This approach is based on the common misconception of equating priority and importance. However, with the calculated worst case execution time available, standard priority-assignment theory can be used [11]. The Metronome [12, 14, 13] collector is another example of a time-based collector. This collector is investigated in detail in Section 2.1.6.

Quantifying the Cost of Incremental Approaches

Irrespective of whether a schedule-based or work-based approach is used, the additional time cost of an incremental approach over a stop-the-world one comes from one primary source: maintaining consistency between the mutator and collector through the execution of barriers. [15] (Space overheads are negligible or zero, at most being an array of pointers in some approaches used as part of the barrier implementation.) Quantifying this overhead is often difficult as the work done at each increment involves the execution of other tasks such as tracing and defragmentation. Since these overheads are being treated independently, the cost of a barrier here is considered only to be the cost of maintaining a suitable consistency between the mutator's view of the object graph and the actual object graph. There is a substantial amount of technical detail involved in the different types of barriers such as whether read or write barriers are used, whether a snapshot-at-the-beginning[16] or incremental update technique[17] is used, the type of read barriers used (Brooks [42] or Baker [16]) and whether read barriers are eager or lazy.[18] Rather than investigating the costs of all these approaches, it suffices to estimate a cost for these barriers. In [127], Zorn shows that the cost of a read barriers alone can incur a 20% penalty on application performance when executed in software. Bacon *et. al.* [14] claim that their Metronome garbage collector can reduce this to 4% on average and 9% in the worst case. Considering a read barrier based on pointer updates can be implemented with a

[15] There is also the extra vertical switching between application and runtime to consider but this is comparatively small.

[16] In this approach, the object graph at the end of the collection cycle could be significantly different from the actual graph but newly created objects do not need to execute a write barrier.

[17] In this approach, the collector's view of reachable data is incrementally updated in the face of changes by the mutator.

[18] Eager barriers maintain the invariant that all references are made to point to to-space immediately whereas lazy barriers do not.

handful of operations (an average four ALU/branch instructions in [127] and a compare, branch and load in [12]), these significant overheads are caused by the large number of times these barriers are executed. Write barriers are executed on what are commonly less frequent writes but are generally more expensive. For the purposes of this investigation, their cost can be approximated to that of read barriers.

2.1.5 The Cost of Defragmentation

The worst case overheads of fragmentation in the DMA motivate a space-time tradeoff for real-time environments irrespective of the fact that the observed fragmentation is often very low. One solution to reducing fragmentation is to carry out runtime *defragmentation* (or *compaction*), a process in which memory is rearranged and compacted. This is only possible for languages without direct addressing into memory. Although defragmentation requires time that again can make a system unschedulable, some DMA algorithms may exhibit worst case memory requirements that make defragmentation a necessity. Assuming a complete defragmentation occurs when an allocation request fails where it would not have in a defragmented system, a direct tradeoff between time and space can be observed. Typically, the heap size would be made a factor of the maximum amount of live memory (for example double the size) and at some trigger point (for example when an allocation fails), the defragmentation routine is invoked. Therefore, the worst case time required is broadly a function of the maximum heap size, the frequency that this size is reached (that is some form of "fragmentation factor")[19] and the cost of a copy. This latter factor depends on the DMA policy and is highly machine dependent. There is very little information on the overhead of defragmentation in explicitly managed memory models, both in the non real-time and real-time domains. The former is because memory defragmentation is rarely implemented due to the infrequent necessity of defragmentation in real-world applications. The latter is because of the traditional focus of minimising time overheads over space overheads in real-time environments. Also, in both cases, explicit memory management is usually implemented with direct access to memory, thereby making it impossible to move data.

Defragmentation often becomes a necessary component of real-time collectors due to the requirement of assuming the worst case space requirements and is further encouraged by the reference abstraction that is already available. Nevertheless, the infrequent occurrence of fragmentation has prompted a move away from the intrinsically defragmenting nature of semi-space copying collectors to mark-sweep style collectors which are "mostly non-copying". Therefore, reducing space requirements through defragmentation still remains secondary to limiting time overheads. An idea of the

[19]It is noted in passing that the amount of fragmentation is elegantly described in terms of a coefficient λ in [14]. The cost of defragmentation can therefore be less conservative that the worst case of moving all live data at maximum fragmentation.

cost of defragmentation for such a collector in real-world applications can be obtained from [14]. The cost of defragmentation ranges from 3% of the collection cycle in the observed worst case of real-world applications to as high as 40% of the collection cycle in the observed worst case of a synthetic high fragmentation application. Space requirements are thereby kept to about twice the amount of maximum live memory.[20]

Defeating Fragmentation Through Non-Contiguous Allocation

An alternative solution to the fragmentation problem is briefly noted here. This solution is to fragment allocation requests to fit available locations which are usually of fixed size [53, 110]. This solution is only possible in environments in which direct memory access is not allowed. The tradeoff here is moved totally from the space dimension of the policy into the time dimension of the mechanism. This incurs time overheads on allocation that can be excessively high. Using fixed sized blocks as is done in [110] reduces this problem but results in greater internal fragmentation. Using a non-contiguous block of memory also impacts memory access time, thereby introducing further unpredictability or high worst case execution times for tasks.

2.1.6 The Metronome Collector

The space and time costs of using existing DMAs in explicit, fine grain memory models has been covered in Sections 2.1.1 and 2.1.2. However, the wide variety of collection technologies has required a more general overview of the overheads in automatic memory models that gives an idea of the cost of adopting this model but perhaps fails to demonstrate what can be achieved using real-world state-of-the art collectors. In particular, it could be argued that some collectors could perform significantly better in some areas but this generally results in a degradation in others. For example semi-space collectors would be ideal for domains with stringent space requirements but the time overheads due to copying are prohibitive if timing requirements are tight. In order to give a better picture of the best real-time developers can currently achieve with real-time collection technology, a investigation into the overheads incurred by the Metronome [12, 14, 13] collector is given next.[21]

The Metronome collector is a time-based collector that uses a best-fit policy implemented with a segregated free list mechanism in its DMA and an incremental mark-sweep collector that defragments when required. The segregated policy implies that internal fragmentation is observed rather than external fragmentation. A read barrier is implemented to ensure moved objects are properly referenced by the application. In motivating the time-based approach of the Metronome collector, Bacon *et. al.* derive analysis that allows a guarantee of the minimum mutator time. For a given time period in the application, the mutator and collector have two properties specified: for the mutator,

[20] There is a caveat here that is discussed in further detail in Section 2.1.6.

[21] The reader is referred to [13] in particular where the authors argue in favour of using Metronome in the RTSJ as opposed to the scoped memory model.

Benchmark	m	s	T_{GC}	T_I	T_R	T_M	T_S	T_D
javac	86	172	2.21	0.001	0.061	1.973	0.137	0.124
db	82	137	2.63	0.001	0.043	2.408	0.148	0.163
jack	82	146	1.73	0.001	0.042	1.076	0.094	0.047
mtrt	80	122	1.59	0.001	0.046	1.386	0.115	0.078
jess	73	126	0.63	0.001	0.186	0.554	0.046	0.031
fragger	72	151	3.20	0.001	0.147	1.700	0.175	1.295

Table 2.1: Time and Space Overheads at 50% Utilisation in Metronome

the allocation rate over a time interval and the maximum live memory usage and for the collector the rate at which memory can be traced. By defining the frequency of invocation of the collector, the memory required for a given utilisation requirement is derived. Alternatively, the maximum available memory is specified and the minimum guaranteed utilisation is derived. Briefly, given the mutator quantum Q_T and the collector quantum C_T the mutator utilisation is trivially $\frac{Q_T}{Q_T+C_T}$. If the allocated memory between t_1 and t_2 is given by $\alpha(t_1, t_2)$, the rate of collection is R and the amount of live data at time t is given by $m(t)$, then it can be shown that *excess* space (excluding fragmentation) of $\alpha(t, t + \frac{m(t)}{R}.\frac{Q_T}{C_T})$ is required. Based on the particularities of the way Metronome operates, this value is trebled.

The results from Metronome are of particular use to this investigation as the cost of each of the four processes of garbage collection are broken down and quantified. Ignoring allocation costs, collection overheads are broken down into the costs of initialisation and termination of the collector (T_I), root scanning (T_R), marking (T_M), sweeping (T_S)and defragmentation (T_D). For a 50% utilisation across the selection of applications from the SPECjvm98 benchmarking suite, the time overheads (in seconds) are reproduced in Table 2.1 where the amount of live memory (m) and the maximum heap size (s) in Mb are also shown. It is interesting to note that the majority of the collection cycle is spent tracing, with the time taken for defragmentation being significant only in the "fragger" application.

By requiring the user to specify information about the pattern of object usage, the space and time overheads are significantly smaller than those shown to hold in the worst case for an explicit model using a DMA. The information that needs to be specified includes the average object size and locality in the size. These parameters are used to reduce the pessimism in the worst case overheads incurred during tracing and defragmentation. Tools for automatically calculating these parameters for fine-tuning garbage collectors are becoming more common [91].

The results from the research for Metronome initially appear promising. The space overheads are as little as two and half times the amount of live memory. A factor λ that specifies the locality of size of objects is introduced in the analysis to bound the amount of fragmentation that can occur.

Although the cost of tracing and using an incremental approach are still significant, Metronome's λ factor appears to address the fragmentation problem for its DMA. This research therefore provides not only a real-time collector with tighter space and time overheads but, more significantly, a solution that can be applied to explicit, fine grain models for use in more resource constrained environments. This could therefore fill the gap between the environments real-time collectors address and those addressed by the static approach. The λ factor allows the developer to capture Johnstone's thesis that fragmentation in real-world systems is negligible. However, there are two problems with this approach: firstly, identifying the λ factor for an application is non trivial; secondly, the analysis does not capture the possible variance of this value during the application's lifetime. Therefore, the chosen λ factor will always be the smallest value during the entire lifetime of the application.

This approach leads to a number of assumptions in the derivation of the worst case space requirements for a given minimum mutator utilisation that inhibit a true calculation of the worst case space requirements. Crucially, the space-time relationship between heap size and utilisation is not well defined due to the interdependence of parameters leading to undesirable recursive functions. In particular, the amount of extra space required depends on the time needed for a collection cycle which in turn depends on the amount of heap space. The chosen or derived heap size is based on an "expansion factor" of the maximum amount of live memory that is chosen to be around 2.5 based on experimentation. In the absence of the pessimistic values that would be obtained from a recursive relationship of space and time overhead, the results given in [12, 14] are essentially observed rather than analytical quantities. Assuming 50% utilisation, and a mutator allocation rate of mMb/s, this value gives the excess space required as $3.2 \times mMb$. If the heap size is given at a large order of magnitude relative to this, then the factor of 2.5 would probably suffice. If this is not the case, then the choice of heap size could be made a function of the maximum allocation rate and T_{GC} rather than the maximum live memory. This could result in the heap size being several orders of magnitude larger than the maximum live memory. For example, combining the results form [12, 14] reveals that for the "javac" application running at 50% utilisation rate, a worst case allocation rate of 258Mb/s and a root scan and mark phase requiring just over 2 seconds could result in space requirements of over 0.5Gb. This would occur if collection starts at the same time as the mutator begins a phase of maximum allocation.

Finally, two other overheads need to be considered: the time costs of executing read barriers and the space and time overheads of the DMA apart from the policy overhead due to fragmentation. Read barriers add a further performance penalty that is estimated to be about 4% on average and 11% in the observed worst case, the latter figure holding for the "javac" application. Deallocation in the DMA can be approximated to the cost of sweeping (that is, T_S in Table 2.1). For the "javac" application, this is around a 6% overhead, a value that is slightly better than would be expected

when considering Puaut's analysis. This is due to the block deallocation that eliminates frequent vertical switching as would occur in an explicit, fine grain model as well as the efficiency of the segregated list policy. The time overheads of allocation in the DMA are not captured well in this work as they are absorbed into the mutator time. With a segregated best-fit policy, these would be expected to be significantly above 50% in the worst case by Puaut's results for an application of average memory utilisation. In reality therefore, a 50% MMU would mean that the application spends more that half of this time executing allocation requests and read barriers.

2.1.7 Going Beyond Fine Grain Models with Random Arrival Patterns

Having investigated the overheads incurred in traditional explicit and automatic fine grain memory models, an evaluation of these models in relation to ($Q1$) on page 2 can be made. From the above investigation, it is clear that the worst case space overheads due to fragmentation are too high. The low occurrence of the worst case makes fragmentation a minimal issue for non real-time applications but the absence of a provable upper bound makes it the most significant source of pessimism in the calculation of space overheads in real-time environments. Defragmentation can reduce the space requirements significantly but the costs of copying adds a significant time overhead. In automatic memory models, the cost of tracing in particular, as well as the cost of executing barriers in an incremental approach add yet further time overheads.

Despite the large range of implementations of explicit and automatic memory models that have been developed over the past four decades, it is evident that no adequate solution for hard real-time applications with tight space and time requirements has been found. This does not appear to have deterred further refinements to these models and their implementations but the improvement in every case is marginal and when significant improvements in either the space or time dimensions are observed, this is often at a significant deterioration of the other. This raises the question whether these models could ever be refined to a suitable extent as well as how these refinements could be identified. In order to address this issue, it is first necessary to understand the relationship between the space and time overheads incurred by these memory models.

The Space-Time Relationship in Memory Models

The space-time tradeoff phenomenon has been investigated in various areas of computer science. The commonality is that an increase is the requirements in one dimension tends to result in a reduction in the requirements of the other. For example, a bubble sort algorithm uses constant memory space but quadratic time in the worst case whereas a binary sort tree needs linear space but linearithmic time. The existence of a tradeoff here is obvious but the relationship between the two is not. A simplistic relationship would be that if m represents the space requirements of the sorting algorithm

and t the time requirements, the function $m \times t$ would give a constant result.[22] Clearly, this is not the case in the previous example. Moreover, there exist all round better algorithms. For example, heapsort combines the good qualities of bubble sort and binary tree sort so that its time requirement is linearithmic, and its memory requirement is constant in the worst case. The complexities of the tradeoff relationship do not end here however. Considering the worst case requirements alone fails to capture the tradeoff in the average and best cases. For example quicksort is a popular sorting algorithm that is extremely fast in practice as it is linearithmic in time in the best and average case. However, it degrades to quadratic time in the worst case, this being when the data is already sorted. Yet another aspect that needs to be considered for some algorithms are the overheads of the algorithm implementation itself. For example, many sorting algorithms are implemented as recursive functions for which stack space needs to be considered as an additional space overhead. Note that there is a parallel here with the policy-mechanism separation already discussed for DMAs. This discussion highlights the problem of simplifying the relationship between space and time overheads in memory models. The existence of a function that relates all relevant attributes (or even identifies them) has not been researched as has been done elsewhere.[23]

It would be attractive to be able to define a similar lower-bound on the space-time product of the explicit memory model that relies solely on a DMA, even if only considering the attributes of the worst case. This would serve two purposes: it would first allow a quantification of how "good" a DMA is and secondly it would motivate alternative memory models outside the simple fine grain memory model if this lower bound were still unsuitable for real-time environments. The derivation of such a lower bound has not been researched and in all probability the complexity of achieving this is much larger than for other domains such as those previously described. One reason for this is that the separation of policy and mechanism adds a further complication to achieving this. A DMA policy has an inherent worst case space requirement as discussed above but the timing requirements are dependent on the mechanism implementing the policy. For example, the best-fit policy is implemented in Puaut's work using three algorithms which all give different worst case space requirements. Furthermore, the search space that results from a chosen policy has an impact on these timing requirements. Therefore, the complexity of deriving a lower-bound function of space-time requirements arises from the tradeoffs being manifested not just across policy but also across implementing mechanisms and also between mechanism and policy. This intricate relationship is not captured well by analysing Robson's and Puaut's work. This is particularly evident by looking at how their worst case expressions are derived. In order to derive the space overheads, Robson showed

[22] It is noted in passing that is some cases, this direct relationship does exist. For example in [75], it is shown experimentally that $m \times t$ is a constant, where m is the size of the heuristics lookup table and t is search time.

[23] For example in search algorithm research, Beame [20] showed that the optimal lower bound for the space time product of a class of search algorithms is $\Omega(n^2)$. Similar observations have been made for other classes of algorithms. For example, Ajtai [9] proved that for a class of programs deciding the Hamming distance problem in time $O(n)$, $O(n \log n)$ space is required.

how the best-fit and worst-fit algorithms can result in a pattern of single word blocks each preceded by a gap of $n-1$ words.[24] Time overheads are derived by Puaut by analysing the search space of a mechanism. Therefore, while there exists a tight relationship between space and time overheads due to both mechanism and policy, the worst cases are derived using orthogonal criteria.

The space-time tradeoff for automatic memory models is even more complex to characterise. Even at the broadest level, the choice between semi-space copying collectors or low copying mark-sweep collectors is often subjective. Some strategy choices are universal however. In particular, defragmentation is assumed to be a necessary part of any real-time collector. Given the orders of magnitude of space required due to fragmentation in the worst case, the extra time required for data copying is generally considered to be worth the overhead. However, there is no proof of this argument being correct that also relates all the other overheads incurred by defragmentation. For example, in Metronome, defragmentation leads to lower space overheads due to fragmentation but a longer collection cycle. A longer collection cycle then results in an increase in space requirements as defragmentation may be occurring at the point of peak allocation.

Motivating a New Memory Model for Real-Time Systems

The current state of affairs poses a significant problem if the next generation of complex real-time and embedded systems are to target the dynamic memory models currently used in traditional applications. In the absence of a proven lower-bound for the space-time tradeoff of fine grain memory models which assume random arrival patterns, two possible research directions exist: invest further in refining these models or derive alternatives that improve upon the static approach without incurring the space and time overheads of these fine grain models or the development complexity of reuse techniques.

The first solution would require research into the strategies of the better performing DMAs and collectors and an application of these strategies whenever possible in the proposed model. The non-triviality of the space-time relationship and the depth of existing research spanning several decades would suggest that the resulting solution would probably provide only marginal improvements on the existing state-of-the-art and not the orders of magnitude required by the application domain targeted in this thesis. This conclusion therefore motivates more than just a new DMA or collector with a slightly modified strategy; rather it necessitates a new memory model.

Rather than seeking the motivation for a new memory model in an argument that current approaches are inadequate, this thesis states a simple hypothesis that relates a theoretic lower bound on the space and time overheads of different memory models. The Entropy Hypothesis introduced in Chapter 3 is based on an intuitive concept that relates the information available about how memory

[24] This pattern is not necessary in buddy algorithms as the system can break down even when a contiguous gap large enough to satisfy a request is available.

is used by an application with this theoretical lower bound. Using this hypothesis and the insight into the primary source of overheads investigated in this chapter, it is argued that a coarse grain memory model can help provide the order of magnitude improvements required. This is largely for two reasons: firstly a coarse grain model can capture implicit object lifetime information that can reduce the worst case space and time overheads of dynamic memory management, even when using an allocation and deallocation policy identical to those described in this section; secondly, specifying lifetime information at a coarse granularity means that it is feasible to leverage the optimisations of offline algorithms by specifying the relative lifetime of regions, thereby deriving an allocation strategy that further reduces these overheads. These two reasons are treated briefly here in order to cover related work. A more detailed investigation of how both object lifetime aggregation and the specification of region lifetime can be used to reduce space and time overheads is carried out in Section 3.3.

The advantage of capturing implicit lifetime information to reduce memory requirements is evident from the large disparity between the observed and analytical worst case fragmentation. Johnstone's thesis that better strategies for DMAs can be achieved by observing regularities in allocation and deallocation patterns can be carried to the real-time domain in which guarantees of space and time requirements are necessary. This requires that patterns that derive low observed fragmentation must be captured in such a way to give deterministic guarantees on the occurrence of this phenomenon. For example, if the phenomenon of locality of size can be deterministically captured in order to guarantee that the allocation and deallocation points of the two smallest objects is known to occur in immediate sequence, then, by Robson's analysis, the worst case overheads are smaller if these objects are grouped together. Therefore, the same DMA algorithms can be used as in the online policies described in this section but fragmentation is reduced in any policy that guarantees to always allocate these two objects adjacent to each other.

The worst case analysis of Robson considers only DMAs that do not take advantage of any available information on the duration for which the memory serviced for each allocation is required. In [94], Naor *et. al.* show that even when it is known at the point of allocation how long the memory is requested for, the algorithms they develop experience worst case memory requirements as bad as those of first-fit derived by Robson. Later, Kalyanasundaram *et. al.* [80] prove that a marginal improvement on these algorithms can be achieved but this is bounded from below by $M(\frac{\log n}{\log \log n})$. These results indicate that knowing the duration of an allocation request when it occurs at runtime is of little use. However, if the duration and arrival of all memory requests are known at compile-time, then an optimal allocation and deallocation strategy can be adopted by finding the ideal location in memory to service each request.

Algorithms that discover an allocation strategy at compile-time are called "offline" algorithms as

opposed to the "online" algorithms described thus far as DMA policies. Since it can be proven that finding an optimal allocation strategy is NP-complete in the strong sense [63, 111], polynomial time algorithms that approximate a non-optimal solution have been devised. This includes an algorithm devised by Kierstead [82] for which the memory requirements in the worst case are a factor of 6 larger than M (or more precisely, the competitive ratio is 6) and more recent work by Gergov [67] which improves this to a factor of 3. Non-optimal offline algorithms are generally described in terms of a graph colouring problem that solves a special case of the strip-packing problem in which the movement of strips is only allowed in one dimension [15, 94]. Each allocation request is described as a triple (s, r, d) where memory is allocated at time r and deallocated at time d for a memory request of size s. These models rely on the assumption that the lifetimes of all requests can be determined exactly. Consideration of the control flow of a program in order to take advantage of sharing of memory that is known to be allocated concurrently (for example memory allocated in two branches of a conditional flow) has also been researched [59] and uses similar algorithms to the offline allocation problem [67]. Note that even though offline algorithms are in essence an approach to adopting a static memory model, the typed nature of the environment considered here calls for a distinction between the static memory model as defined above (that is where memory is allocated at a pre-mission phase) and one in which the same memory can be used for more than one type of data. In practice therefore, even when an offline algorithm is adopted, a DMA is still required to initialise memory for an allocation request. Such DMAs trade the mechanism-related memory overheads of maintaining the free/busy data structures in DMAs with online policies to a data structure storing the location in memory where every allocation must occur. DMAs that use offline algorithms to derive a memory location for each memory request also incur less time overheads than an online DMA as no time is required at runtime to search data structures for a suitable memory block. Instead, an allocation request requires only looking up where the memory for that request is to be allocated and then initialising that memory.

2.2 The RTSJ and its Memory Model

The RTSJ is still a relatively recent specification, having seen its first release at the end of 2001. Developed under Sun Microsystem's Java Community Process and with the support of leading members of industry and academia, a first version of the RTSJ was available within three years from the start of the specification request. The quoted goal of the RTSJ is that of: *"... providing an Application Programming Interface (API) that will enable the creation, verification, analysis, execution, and management of Java threads whose correctness conditions include timeliness constraints."* A set of guiding principles drove the development of the RTSJ, the key one relevant to the success of the RTSJ being that predictable execution is held in priority in all tradeoffs. Another important

guideline was that no syntactic changes were made to the language, thereby ensuring compatibility with existing compilers.[25] The only changes allowed to achieve the RTSJ's goals were therefore modifications, restrictions and extensions to the virtual machine or equivalent runtime that would implement the RTSJ API. Labelled the "seven enhanced areas," these include:

- the definition of real-time thread scheduling and dispatching mechanisms,
- appropriate synchronisation and resource sharing mechanisms,
- asynchronous event handling mechanisms,
- mechanisms for asynchronous transfer of control (ATC) and thread termination that maintain data integrity,[26]
- support for application access to physical memory and
- *a new memory model that allows the application to avoid the penalties of garbage collection.*

Schedulable Objects and RTSJ Memory Areas

As the heart of the RTSJ is the `Schedulable` object, the unit of execution that the real-time scheduler understands. This added abstraction over standard threads allows other classes such as event handlers to form part of the scheduling and dispatching decision. `Schedulable` objects implement the interface `Schedulable` which, as an extension of the standard `Runnable` interface, defines the abstract method *run()*. The information supplied by the methods defined in `Schedulable` is used by the scheduler to create a suitable context to execute *run()*. The RTSJ defines two types of `Schedulable` objects: `AsyncEventHandler` and `RealtimeThread`. `NoHeapRealtimeThread` extends `RealtimeThread` and implements the same methods but has special memory characteristics as described below. For the purposes of this thesis only `RealtimeThread` and `NoHeapRealtimeThread` will be considered as the semantics for memory areas that are defined for real-time threads apply in the same way to instances of `AsyncEventHandler`.

The RTSJ provides three types of memory areas: scoped memory, immortal memory and heap memory which are encapsulated in the `ScopedMemory`, `ImmortalMemory` and `HeapMemory` classes

[25] In addition to these two guidelines, five other guidelines drove the development of the specification. These are that the specification:

- should address current real-time system practice as well as allow future implementations to include advanced features,
- must not be applicable to any particular Java environment,
- must maintain backward compatibility with non real-time programs,
- must maintain the WORA philosophy where possible and
- must allow variation in implementations.

[26] These are two separate items that are often grouped together as is done here.

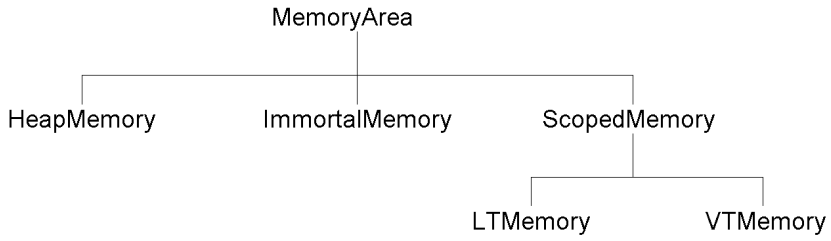


Figure 2.3: The RTSJ Hierarchy of Memory Classes

respectively. The class hierarchy for RTSJ memory areas is shown in Figure 2.3. `HeapMemory` provides application-level access to the standard Java heap in which objects are allocated by `java.lang.Thread` threads. Every application contains one instance of `ImmortalMemory` in which residing objects live for the duration of the application. Every `ScopedMemory` instance is associated with a physical memory area called the *backing store* that can be allocated and freed dynamically. `ScopedMemory` is an abstract class with two subclasses: `VTMemory` and `LTMemory` that describe memory areas with variable and linear allocation times respectively. `RealtimeThread`s can access objects in the heap, scoped and immortal memory areas whereas `NoHeapRealtimeThread`s can only access objects in scoped and immortal memory. When accessing heap objects, `RealtimeThread` instances are subject to garbage collection latencies. By avoiding the heap, `RealtimeThread`s will be free from these latencies as they will always preempt the garbage collector. For the purpose of this thesis it is assumed that no heap access is allowed and therefore only the semantics of `NoHeapRealtimeThread` are relevant. When using the term "thread" it is henceforth assumed that an instance of `NoHeapRealtimeThread` is being considered.

The Semantics of RTSJ Memory Classes and their Methods

Every RTSJ thread executes in some memory area at any time and this memory area is called the *current allocation context* of that thread. When a thread invokes the `new` operator to create a new object, that object is created in the current allocation context of that thread. For example, if the current allocation context of a thread is the immortal memory region, then any new objects created are allocated from the immortal area. By changing their current allocation context, threads can create objects in either immortal or scoped memory regions. Changing allocation context is done using the methods of the `ScopedMemory` class. Allocation and deallocation of scoped memory regions is defined by the semantics of scoped memory as defined in the RTSJ. An overview of these semantics is described next using an example. This description is not complete but suffices for the purposes of this thesis. For example, there are several ways to change the current allocation context of a thread that are not described here. Further details are given as necessary when investigating

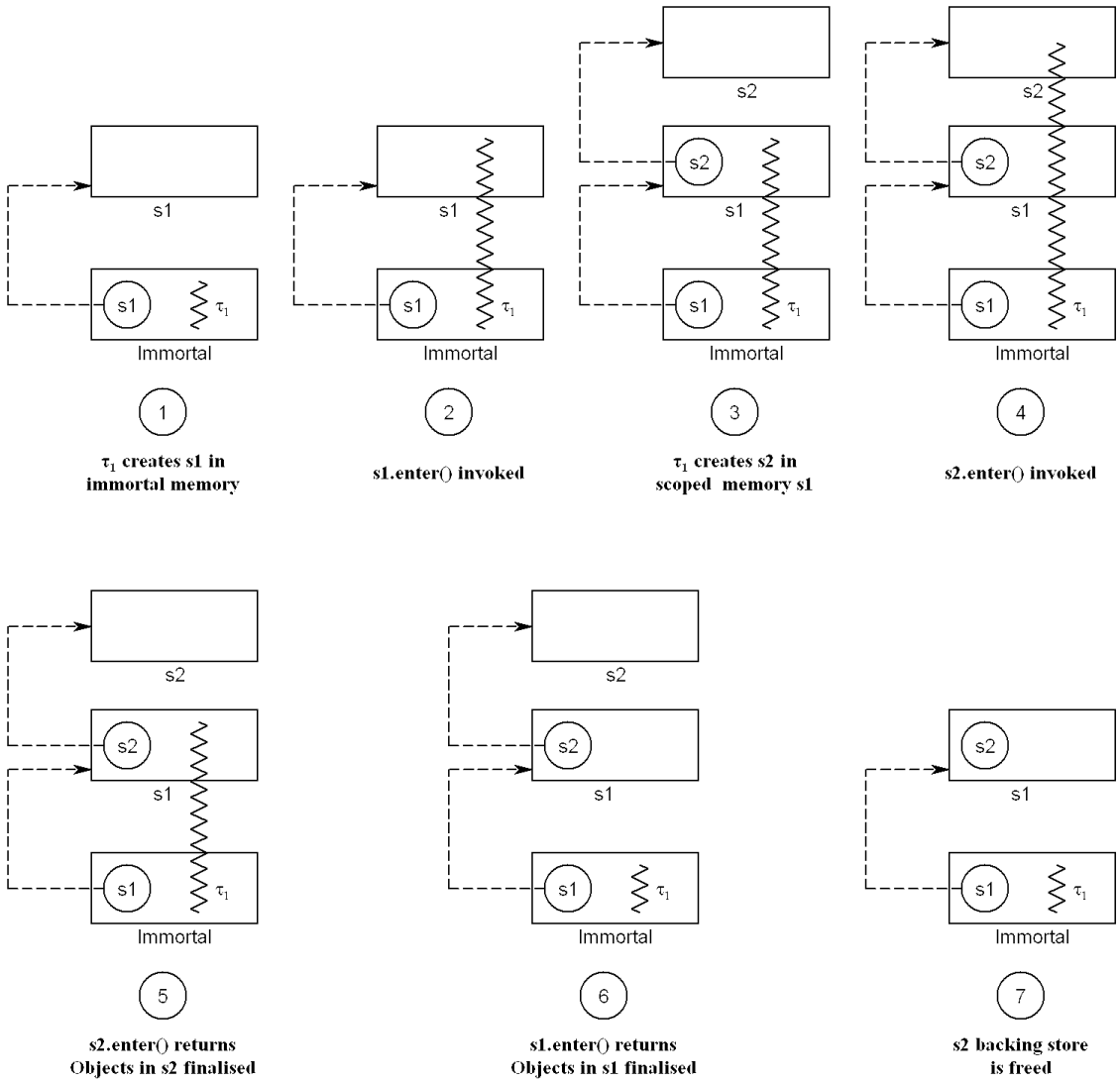


Figure 2.4: Changing Allocation Context Using *enter()*

particular parts of the memory model.

For most purposes, it can be assumed that the initial allocation context of a thread is immortal memory. Let τ_1 be a `NoHeapRealtimeThread` executing in this context. Therefore, all objects created by τ_1 are allocated from immortal memory. If τ_1 creates a new instance of a realisation subclass of `ScopedMemory` (s_1), then this object is also created in immortal memory. This arrangement is shown in Step 1 in Figure 2.4. Associated with s_1 is the backing store for this memory area. This is the physical memory block allocated from memory that is used for allocated objects for which s_1 is the current allocation context. By invoking the *enter()* method of `ScopedMemory` as shown in Step 2, the current allocation context changes to s_1 and the *run()* method of the `Runnable` instance of s_1 begins to be executed. All objects created by τ_1 are now created in s_1. Similarly, τ_1 can

create a `ScopedMemory` object s_2 in s_1 (Step 3) and change the current allocation context to s_2 (Step 4). When a thread changes its allocation context to a new memory region using *enter()*, that thread is said to "enter that region". The regions more deeply nested in the scope stack are called *inner regions* whereas those less deeply nested are called the *outer regions*. For example, s_2 is an inner region with respect to s_1 and the immortal region is outer to all scoped regions. Note that this relationship depends on the structure of the backing stores and not the location in which `ScopedMemory` objects are created.

When *run()* returns in s_2 at the completion of execution of this method, *enter()* also returns (Step 5). At this point, the current allocation context of τ_1 is reverted back to s_1 and all objects in s_2 can be finalised. s_2 is cleared of all objects at this point but its backing store is still present in memory. Similarly, when *run()* returns in s_1, the current allocation context becomes immortal memory and all objects in s_1 are finalised, including the object s_2 (Step 6). Once this object is finalised (and no other `Schedulable` objects are associated with this region as explained below), the backing store for s_2 can be deallocated in the runtime.

A second method that can be used to change the current allocation context of a thread is the *executeInArea()* method of the `ScopedMemory` class. Consider Step 1 of Figure 2.5 which shows a thread with the current allocation context being s_2. (This step is the same as Step 4 in Figure 2.4). By invoking the *executeInArea()* method of s_1, the allocation context is changed to s_1 (Step 2). *executeInArea()* takes a `Runnable` instance as a parameter and the *run()* method of this object is executed. When *run()* returns, *executeInArea()* also returns and the allocation context is changed back to s_2 (Step 3). *executeInArea()* can be used to change the allocation context to any memory region below the current allocation context in the scope stack. This is shown in Step 4 where the allocation context is changed to immortal memory. *enter()* can be invoked at any time to change the context to memory areas both in the existing scope stack as shown in Step 4 (and subject to the single-parent rule that will be explained shortly) or in a new memory region as shown in Step 5. This results in a cactus-like architecture of memory regions whereby outer regions are guaranteed to live longer than inner regions and regions closer to the tail of the executing thread live longer than those at the head.[27]

Scoped References and the Single-Parent Rule

A scoped memory region can be the current allocation context for more than a single thread at any instant. In order to deterministically guarantee that no region is deallocated while a thread can still access that region, each region maintains a *reference count*[28] of the number of `Schedulable`s objects associated with it. When a thread enters a scoped memory area, this count is incremented

[27] As discussed later, the second part of this statement is true only of single-threaded environments.

[28] This is a count of the number of `Schedulable` objects associated with this region and should not be confused with reference-counting as applied to objects in reference-counting garbage collectors.

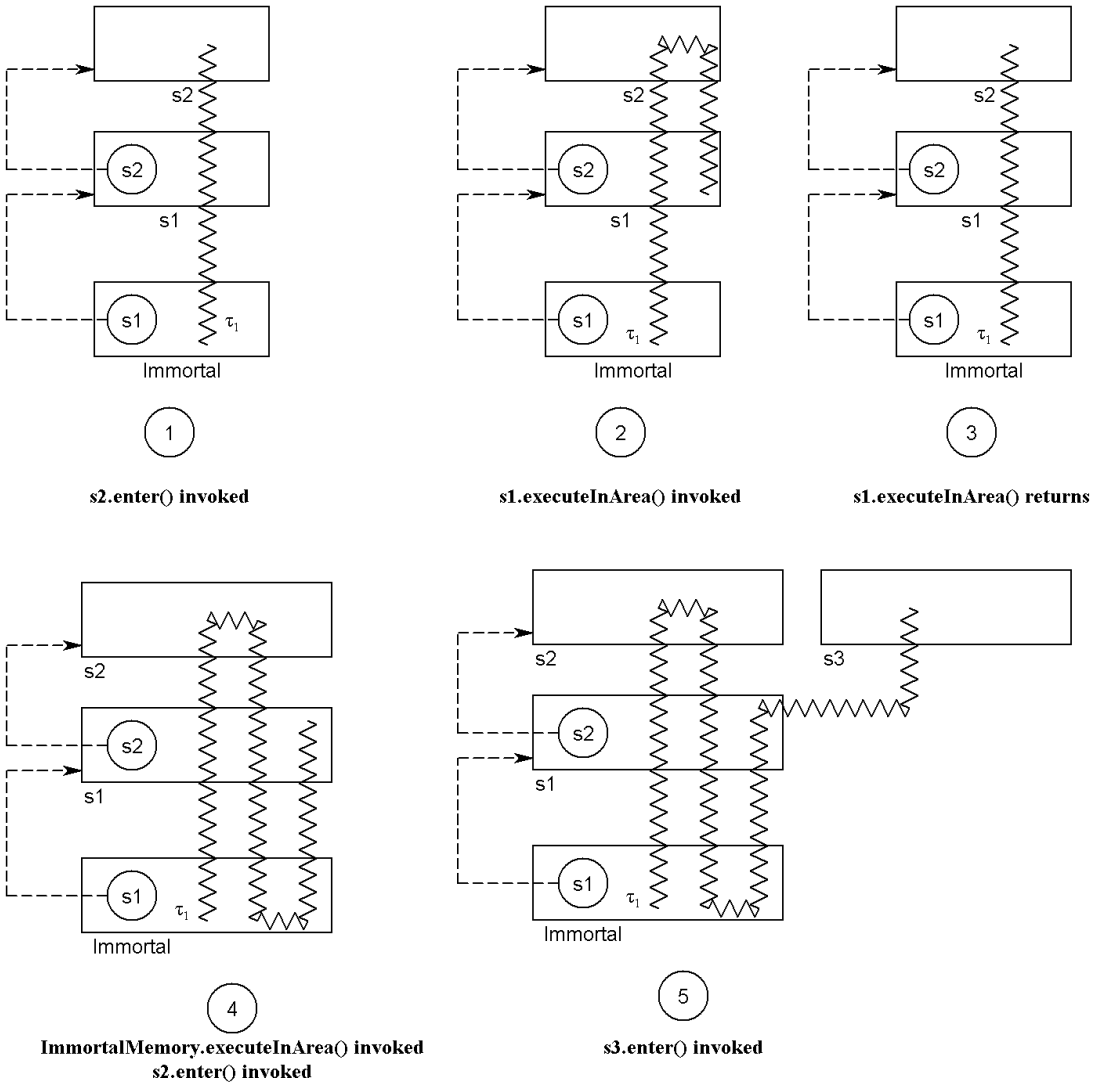


Figure 2.5: Changing Allocation Context Using *executeInArea()*

and when *enter()* returns, it is decremented. When the count falls to zero, all objects are marked for finalisation and when the `ScopedMemory` instance representing that memory area is also finalised, the backing store can be deallocated. In order to reduce the complexity of RTSJ implementations, the RTSJ requires threads to obey the "single-parent rule". This dictates that scoped memory areas are to have only one parent in the entire system across all threads. Once a memory area is entered, its parent is recorded and set to the previous memory area context; the combination of scope stacks from the threads in the system making up the scope tree. Since every thread can have only one parent, no cycles in the scope tree are allowed. These constraints are typically checked at runtime and a `ScopedCycleException` is thrown if the single-parent rule is broken.

Figure 2.6 shows how two threads might share memory regions. In Step 1, two threads are started with the initial allocation context being immortal memory. Thread τ_1 enters two memory regions in turn, the reference count being incremented to 1 in each case (Step 2). In Step 3, τ_2 enters memory region s_1 and the reference count for this region is incremented to 2. Next, τ_2 enters a new memory region s_3, creating a cactus stack architecture of the memory regions (Step 4). If τ_2 now attempts to enter s_2, the single parent rule is broken as s_2 would have two parents, s_1 and s_3. Therefore, in Step 5 a `ScopedCycleException` is thrown. In Step 6, the s_2.*enter()* and s_1.*enter()* invocations return in τ_1 and the reference count of s_2 is set to zero whilst that of s_1 is decremented to 1. The backing store for s_2 can now be cleared and the parent for s_2 set to `null`. As shown in Step 6, τ_2 can now legally enter s_2.

The RTSJ Reference Rules

The introduction of a dynamic memory model controlled by the user introduces the possibility of dangling references occurring. For example, if an object in immortal memory references an object in a scoped memory area, that scoped memory region could be deallocated during the application's lifetime. The RTSJ's solution to this problem is to require that if some object o_1 holds a reference to some object o_2, then the lifetime of the memory area holding o_1 must live at least as long as o_2. The single-parent rule always guarantees that inner memory areas live less than outer memory areas and the reference rule follows: object o_1 in memory area μ_1 can hold a references to object o_2 in memory area μ_2 if $\mu_1 = \mu_2$, or μ_2 is an inner region with respect to μ_1. As use of the heap is not allowed, immortal memory can be considered to be the root of all scope stacks and therefore all objects can reference objects in immortal memory but objects in immortal memory can only reference objects that are also in immortal memory. The RTSJ reference rules are summarised in Table 2.2.

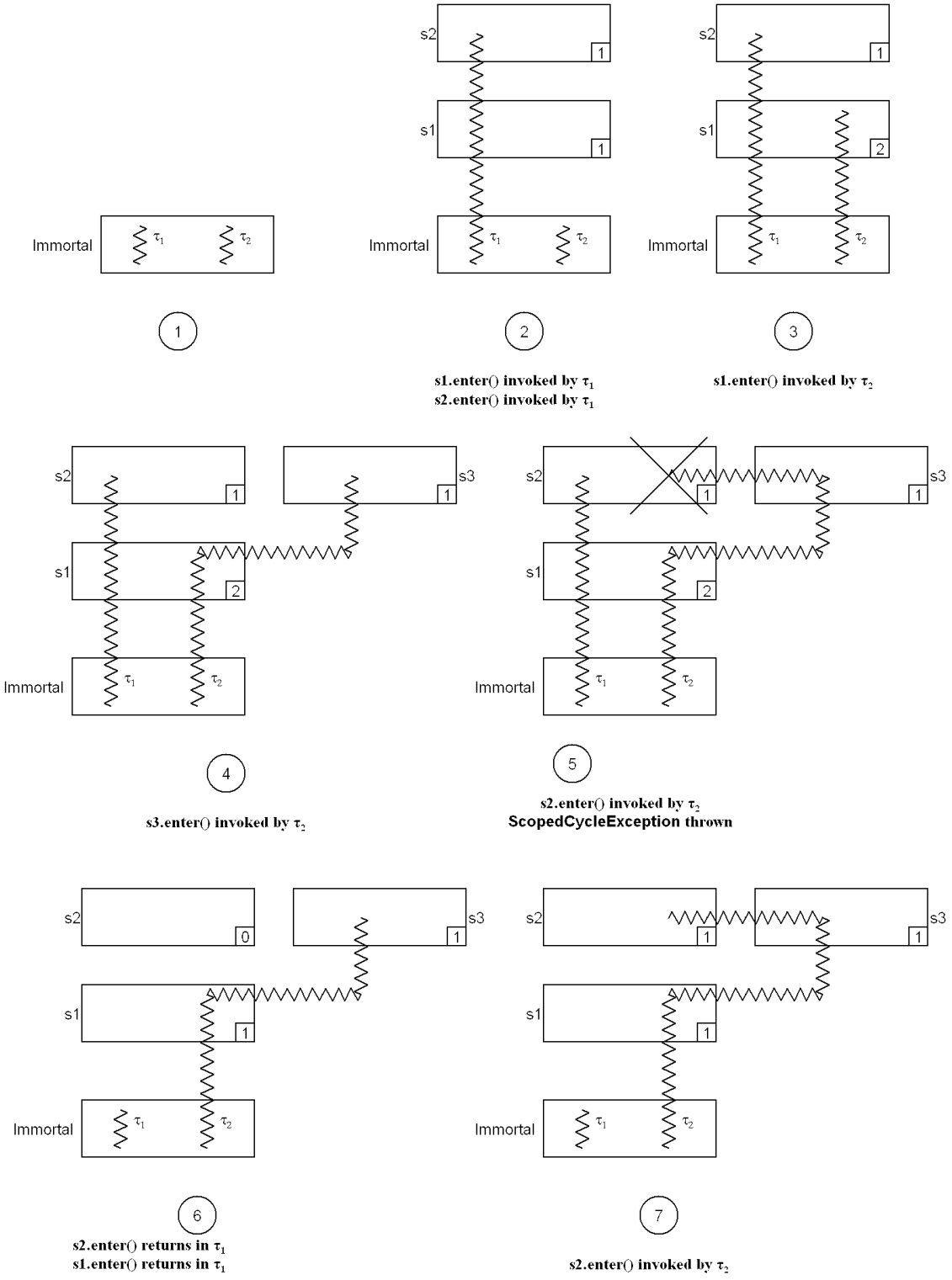


Figure 2.6: Sharing Memory Regions in the RTSJ

Stored In	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Permit	Permit	Forbid
Immortal	Permit	Permit	Forbid
Scoped	Permit	Permit	Permit, if the reference is from the same scope or an outer scope
Local Variable	Permit	Permit	Permit

Table 2.2: The RTSJ Reference Rules

2.3 Investigating Coarse Grain Memory and the RTSJ Approach

Several solutions attempting to alleviate the complexity of developing real-time software using the RTSJ memory model have been proposed. Section 2.3.1 investigates this work and the rationale behind the approaches taken. A lesser amount of work has also been devoted to investigating implementations of the model and their real-time performance. This work is described briefly in Section 2.3.2. Section 2.3.3 gives an overview of case studies which have used the RTSJ memory model, either as an investigation into its advantages and disadvantages or to implement real-time applications. Finally, Section 2.3.4 describes other work that is relevant to the coarse grain memory model and the scoped approach.

2.3.1 Addressing the Complexity of the RTSJ's Memory Model

The primary sources of development and maintenance complexity in using the RTSJ's memory model comes from the single-parent and scoped reference rules. The lifetime of regions is defined and embedded in code and the object reference graph must be precisely specified in order to prevent exceptions. This not only raises problems for developing and maintaining new software but also means that existing libraries can not be used without some specification of the way their methods manipulate references and use scopes. Even with full knowledge of the object reference graph and the lifetime of objects, the scoped model as presented in the RTSJ is restrictive to the extent that common patterns of memory usage are incommensurably complex to implement. For example, implementing thread groups in the presence of scope-using `RealtimeThread`s would require the creation of references that would break the RTSJ rules. The RTSJ chooses to ignore this problem by eliminating thread groups completely.

One approach to eliminating this complexity is to automate the derivation of scopes and the objects assigned to these regions. This not only simplifies development and maintenance but also allows the use of existing libraries. In [55], Deters *et. al.* describe such an algorithm that derives

the scope assignment based on object lifetime and the reference graph. This information is obtained through program tracing [56] and therefore needs to be supplemented with a more conservative static analysis. The authors describe how the reference graph need not be a single dynamic instance of an object but a more coarse grain entity such as a class. They opt to use allocation sites as the nodes of the reference graph so that all objects created at a *new()* site are created in the same scope. Once a reference graph has been derived and the lifetime of objects identified, the program is instrumented to assign scopes that satisfy the lifetime requirements of objects and the reference requirements of the scoped model. It is accepted by the authors that the work described in [55] requires further investigation, particularly into how information from traces can be married with static analysis. Extending the annotation process to consider multi-threaded environments is also yet to be addressed. A similar approach is adopted in [62] but with the use of static analysis in order to capture which objects escape a method. The scope assignment is made sensitive to the call graph rather than the program's allocation sites, thereby being more accurate but requiring more runtime information to be maintained.

The implementation of common patterns of memory usage in the RTSJ memory model is addressed by Benowitz *et. al.* [24], Pizlo *et. al.* [99] and Corsaro *et. al.* [52]. One solution is to map well known software patterns to the RTSJ scoped model. For example, the factory pattern [61] can be extended to take a `MemoryArea` instance as a parameter for every method in the class so that every invocation can specify the memory area which is to be made its allocation context. Corsaro extends the leader-follower pattern [107] to operate in an RTSJ environment using real-time threads and scoped memory. Another solution proposed by Benowitz is to map alternative memory models on top of the RTSJ. For example, a memory pool model is introduced where each object can be recycled from a pool of objects of its class type in immortal memory. He also proposes an explicit, fine grain memory model implemented as a stream of bytes in immortal memory. Mapping alternative memory models to the RTSJ is of little advantage in terms of space and time performance as the disadvantages of these models are only supplemented by the extra cost of maintaining this abstraction. A result of the novelty of the RTSJ memory model is that the best way to *target*[29] the model is unknown. Pizlo proposes a number of new software patterns specifically aimed at the RTSJ memory model. For example, he describes the "multi-scoped object pattern" that allows an object to express which of its methods are safe to invoke within any memory area. This is tantamount to a per-method specification of how object references are manipulated and memory areas used. Other patterns attempt to circumvent the RTSJ rules completely. For example "wedge threads" are used to keep a scoped region alive when its reference count would have otherwise dropped to zero. A "handoff pattern" is also proposed that uses object copying to allow objects to move between

[29]The importance of targeting a memory model is discussed further in Section 3.2.

regions. Although the introduction of these patterns highlights the complexity of using the scoped memory model, forcing a solution on top of the existing RTSJ model has earned them the term "anti-patterns" [52]. There are two possible conclusions that can be drawn: either the RTSJ needs to be extended to allow these patterns to be expressed as an integral part of the model or the way the model should be targeted is still not understood. These arguments are revisited in Section 3.2.

One approach to address the development complexity of the RTSJ memory model is to separate memory concerns from functional code. In [85], Kwon *et. al.* describe how an annotation of methods can be used to specify that a new allocation context is required for an invocation. The granularity of a memory region is therefore one or more method invocations. Amongst other parameters, the annotations specify the size of the region that needs to be allocated and the object that is returned after invocation. As no explicit allocation of scopes occurs, the single-parent rule does not need to be checked. This model is equivalent to a subset of the RTSJ with only an *enter()* method available and that copies the returned object of an invocation to an outer region of the scope stack. As a subset of the RTSJ, this approach still fails to allow common patterns of memory usage to be expressed. The advantage of this approach to developers is therefore primarily in the separation provided by the annotations. The state-of-the-art for separating cross-cutting concerns is Aspect Oriented Programming (AOP) [60]. The only work addressing how AOP can be applied to the RTSJ memory model is described in [119]. The example provided is however limited to specifying allocation as a joinpoint for which advice specifies the allocation context.

Boyapati *et. al.* propose using a system that merges region types with ownership types [41]. Region types define a lifetime relationship of regions to ensure that programs using region-based memory never follow dangling references. Ownership describes an aggregation relationship so that if a type is created in a region r and "owns" another type, then objects of both types are created in r. The ownership relationship therefore dictates the nesting order of regions as well as the objects that are allocated in those regions. The "outlives" relationship from the region type will validate or invalidate pointer accesses. Using annotations to express these relationships, the scoped structure and object assignment to scopes is inferred. It can therefore be guaranteed statically that an application that is properly typed by the rules of the type system will not fail on a reference check or scoped cycle exception. This provides a significant advantage over expensive runtime checks that throw unchecked exceptions. The approach taken in [41] is to provide a per-allocation-site specification of ownership and therefore has the disadvantage of requiring re-writes if the allocation context changes dynamically.[30] The ownership relationship enforces an aggregation that maps well to the scoped structure of memory areas. However, this aggregation is being enforced on the association relationship of objects instead of on the phases of the lifetime of an executing `Schedulable`.

[30]Note however that this could easily be addressed by allowing dynamic instances to be expressed in the type system.

Zhao *et. al.* propose an approach similar to Boyapati's in that static information is used to guarantee the absence of scoped cycles and dangling references [126]. Rather than using explicit annotations, the scoped structure of standard Java packages provides the structure of the scope tree as well as the allocation context for objects. Classes defined in a package can be annotated as "gate" or "scoped", where an instantiation of the former creates a new scoped region in which instances of the latter are allocated. A type system is developed that uses rules to ensure proper nesting (for example only one gate class can exist in a package) and guarantee pointer safety (mainly by specifying that scoped classes can only reference objects in the same package). In contrast to Boyapati's approach, the explicit scope ordering using packages rather than ownership annotations provides a less flexible development environment which also limits polymorphism. The guarantee against dangling pointers is similarly achieved through the specification of an aggregation relationship; however this is not done at allocation sites but at the level of the class and package specifications.

2.3.2 Implementing the RTSJ's Memory Model

The first implementation of the RTSJ was jTime, the reference implementation provided by TimeSys Corporation [1]. In [96] the cost of using scoped memory in this implementation is demonstrated graphically by comparing the application utilisation when using garbage collection with that of an equivalent program using scoped memory. The garbage collected version of the application exhibits utilisation spikes requiring nearly 70% of the processor. When using scoped memory in the same application, these spikes are replaced with much smaller spikes that appear when a scoped region is pushed or popped. The baseline processing cost increases marginally due to the cost of executing the reference checks. It is noted in passing that the cost of reference checks will depend on the number of reference store operations in the application, a value that is known to average about 5% of all executed instructions [73]. The cost of this check[31] is minimal; in jRate it is implemented with at most two integer comparisons [51].[32] The marginal increase in processor utilisation in [96] corroborates this argument even in the less than optimal implementation of jTime. However, the results of Beebee [21] for the FLEX [22] implementation of scoped memory show that reference checks add a significant overhead compared to an environment with checks being disabled, even if there is still a significant performance improvement over using garbage collection. The reason for this result probably lies in the fact that the memory model for FLEX is itself written in Java. Nevertheless, both Niessner's and Beebee's results justify the use of scoped memory over garbage collection for better performance, at least for the applications under investigation. However, in neither case is the space overhead of using scoped memory considered.

[31] The literature sometimes refers to this check as a write-barrier, perhaps giving the false impression that this check is as expensive as the write-barrier of garbage collectors.

[32] If these overheads are still enough to make a system unschedulable, then it is possible to invest in static analysis to eliminate or reduce these checks.

The jRate RTSJ implementation [54, 50] improves the cost of executing reference checks from linear in scope depth as proposed in the RTSJ specification to a constant time algorithm [51]. This is achieved by recording for each region the set of valid regions for which an object in the former can reference an object in one of the latter. The number of regions is therefore limited by the size of the data structure holding this information. A new class called `CTMemory` that extends `ScopedMemory` is implemented that provides constant time object allocation by initialising memory when the reference count goes from zero to one [50].

The first full implementation of the RTSJ outside the reference implementation that will have industry backing is under development at Sun Microsystems. Called Mackinac [6] this virtual machine is yet to become commercially available and there is little information available about the performance of the implementation. Preliminary experiments show that Mackinac is 2% to 40% slower than when executing the same application on the basic HotSpot virtual machine on which it is built. It is not stated what component of this overhead comes from the memory model.

2.3.3 Experience in Using the RTSJ Memory Model

There exists little documentation on real-world applications that adopt the RTSJ memory model. Experience of using the model are typically found in the literature as passing remarks; for example Beebee and Rinard state in [22] that they found it "close to impossible" to implement some programs. The complexities of using the RTSJ are best demonstrated in the examples used to motivate the patterns described at the beginning of Section 2.3.1. The examples here highlight the patterns of memory usage in which the RTSJ proves most restrictive.

Benowitz and Niessner successfully port flight software used on a space platform and originally implemented in C to the RTSJ [25]. They claim to have experienced some difficulty using the RTSJ memory model that can be facilitated by using factories for which the allocation context is expressed explicitly at instantiation.

Basanta-Val *et. al.* use scoped memory in an RMI runtime to clear objects after a remote invocation is handled [19]. Krishna *et. al.* adopt a similar approach in a real-time CORBA core [84]. Scoped memory suits the lifetime ordering of remote invocation as proposed here but the RTSJ reference restrictions means that deserialised parameters and the returned object can not be used after invocation.

Bollella *et. al.* describe the use of the RTSJ memory model in the Golden Gate project [2]. They describe how a memory pool model can be built on top of the RTSJ by preallocating objects in a region and then retrieving objects from the pool and returning them when they are no longer required [30]. As argued earlier, there can be little motivation for implementing an alternative memory model on top of the RTSJ's model. The authors acknowledge this, stating that this approach

can lead to memory leaks and corrupt data if an object is prematurely returned to the pool. A scratchpad pattern is also described to create temporary data that is then shared through copying. The result is described by the authors as a flexible model that manages the RTSJ's complexity. It could therefore be argued that for this application, an object pool model and explicit deallocation would be a better memory model. It is of some concern that the only major undertaking of an application using the RTSJ memory model to date has been addressed by mapping this model to an alternative one.

2.3.4 From Region-Based Memory Management to the RTSJ

Scoped region-based memory management was first proposed by Tofte and Talpin in their 1994 seminal paper [117] and is described in more detail in [118, 116]. The proposed model was successfully implemented in the ML Kit with regions. Briefly, region analysis is carried out using a type-effect system to automatically infer the points of region allocation and deallocation and assign the values of expressions to regions. The results are expressed using annotations that define the allocation context of ML expressions. The inference is in essence an escape analysis of the nested expressions and neatly fits the declarative nature of functional languages. Nevertheless, Christiansen and Velschow show how this approach can be adopted in a subset of Java through a similar inference system that also considers object-oriented features [49].

Chin *et. al.* propose a similar approach to Tofte and Talpin's but target a subset of the Java language [47]. Their approach is clearly adaptable to the scoped model of the RTSJ. Briefly, the initial allocation context for objects is derived by using the fields of classes so that if an object holds a field to another object then the latter will be allocated in the same or an outer scope. Extra region constraints are then derived by carrying out a procedural flow analysis on methods. In this way, it differs from Boyapati's work [41] in that region annotations are derived automatically. This approach is more conservative than Tofte and Talpin's as the lifetime of objects is influenced by the direction of reference. It is basically an example of adopting escape analysis to derive object lifetime and is therefore subject to the same undecidability problem [57].

In addition to the scoped approach to coarse grain memory management, there have been several proposals for explicit, coarse grain allocation and deallocation in arbitrary order. These include Ross' "zones" as early as 1967 [106] and Hanson's "arenas" in 1990 [70] and Berger's "reaps" in 2002 [26]. Gay and Aiken [64] propose this approach and show that even with the additional cost of checking that a region can safely be deallocated, explicit regions perform better than both explicit and automatic fine grain memory management. With annotations expressing known information about references and a supporting type system to check these annotations, a further improvement can be obtained [65]. In an earlier work, Aiken *et. al.* use the annotation results from Tofte and

Talpin's inference system to derive an explicit, coarse grain model by deallocating regions earlier and allocating them later than the original lexical scoping would require [8]. This therefore maintains the safety of Tofte and Talpin's type system while potentially decreasing the application's memory requirements.

The RTSJ opts for a coarse grain model as a tradeoff between the static and fine grain memory models in the spirit of Gay and Aiken. The motivation for a scoped approach to this model is primarily to define a set of rules against the ordering of regions which, if adhered to, guarantee the absence of dangling references. This is a subtle but important difference from the motivation of the inference systems described above for which region nesting serves mainly as a framework for capturing escape information. Nevertheless, both automatic inference as in Tofte and Talpin and correctness checking through annotations as in Boyapati's work could be of benefit to RTSJ application development. This thesis is compatible with this work as relative object lifetime is required as input to the developed algorithms. However, the focus will be on the implications of the scoped, coarse grain model on space and time requirements of real-time systems, an area not covered by this work.

2.4 Summary

This chapter has presented past research into the space and time overheads of explicit and automatic fine grain memory models. A detailed investigation was carried out, in particular on models that do not take advantage of any known information about the lifetime of objects as this is the most common approach adopted in traditional application domains. In these models, the space overhead due to fragmentation was shown to be several orders of magnitude larger than the maximum amount of live memory in the worst case and much larger than that typically observed at runtime. Memory operations are computationally expensive and account for a large portion of processor utilisation for a typical application in the analytical worst case, even with the best performing DMA. Garbage collection will always perform worse than a fine grain explicit model when assuming the abstraction introduced in Figure 2.2 as tracing is required to identify dead objects. Incremental collection adds yet a further overhead due to the consistency requirements when executing the collector and mutator concurrently. The scoped memory model of the RTSJ was introduced together with a description of relevant work that addresses this model. Finally, relevant work on coarse grain models carried out outside the context of the RTSJ was described.

As the overheads of current approaches to fine grain memory management have been demonstrated to be unsuitable for the real-time systems with tight space and time requirements, an alternative approach is necessary. Although the advantages of a coarse grain model may be intuitive, the impact of adopting a scoped approach is not. Therefore, making the case for the necessity of a coarse

grain model to target a key niche of real-time and embedded applications does not automatically justify the approach taken by the RTSJ. The implications of using this model on the space and time overheads of the application are unclear. Moreover, while it is justifiable that a user-controlled memory model would incur higher development costs, in the RTSJ these are sometimes incommensurate with the advantages of a dynamic memory model. The next chapter will formulate a hypothesis that attempts to formalise the relationship between the space, time and development costs of an application and characterise this relationship for static, fine grain and coarse grain models. This is used to motivate a detailed investigation into where the benefits of the scoped, coarse grain model lies and how these benefits can be exploited without the problems experienced in the RTSJ.

Chapter 3

Motivating a Scoped, Coarse Grain Memory Model

In motivating a suitable memory model for hard real-time Java applications as relevant to the questions posed on page 2[1], this chapter introduces a simple way of comparing different memory models. It is argued that despite the complexities of deriving a characterisation of space-time tradeoffs or their combined lower bound, a relationship can be shown to exist between these two dimensions and an additional dimension, "development costs", across all memory models. Development costs are manifested in the expression of known semantic information about the way an application makes use of memory. Therefore, without having to provide a complete characterisation of the relationship across these three dimensions, it can be argued that higher development costs in expressing this semantic information become inevitable in order to achieve the order of magnitude improvement required over traditional fine grain models. The direction of research therefore changes from a fine-tuning of existing solutions that simply shift overheads between the space and time dimensions to devising ways that allow an increase in the developer's ability to express memory usage information to the point that the space and time requirements of the system are satisfied.

3.1 The Evaluation Metric

Proving that a memory model can achieve the goals outlined in the first part of the thesis hypothesis requires an evaluation of the different available memory models. This evaluation must be defined against some metric which measures a set of parameters and can be used to understand the classes of applications that can benefit from a particular model over other solutions. The metric chosen shall be called the *Evaluation Metric* for memory models and consists of three parameters:

[1] To reiterate, these questions were:

(Q1): What is the best memory model to use in a particular class of real-time systems?

(Q2): What is the best memory model to use given the syntax and semantics of the Java Language and Virtual Machine Specifications?

- time overheads,
- space overheads and
- development cost.

In Section 2.1.7 the relationship between space and time was shown to be significantly complex but, in general is described by a function in which an increase in overheads in one dimension tends to result in a decrease in the other. The choice of whether to use a defragmentation algorithm is an example of this. The third parameter introduced in this evaluation metric captures the burden placed on the developer in describing the known information about how objects are used in the application; this concept will shortly be developed further. Development cost is rarely qualified in the traditional fine grain approaches introduced in Section 2.1, in all probability because explicit and automatic approaches at this granularity describe two extremes that are easy to characterise. In this chapter, it is argued that this burden as captured by the expression of this information is not independent of the space and time dimensions; rather it defines them. In particular, development cost shall be shown to be inversely related to both time and space overheads in general; that is, capturing more information about memory usage could decrease both time and space overheads. Introducing development cost as a third component of this relationship does not simplify this relationship but does direct research towards a new direction beyond fine tuning existing models: developing techniques in which additional development costs are indeed translated into all-round better performing models.

In order to describe this tri-dimensional relationship, it is first necessary to understand in more detail what is meant by the term "development cost". All three components of this relationship are related solely to the overheads of the memory model and not to those of the application. Therefore, space and time overheads are defined in a similar fashion as in the previous chapter. Development overheads are also attributed to the memory management concern and not to the development complexity of the entire application. Unfortunately, although time and space overheads are easily measured, the development cost of targeting a memory model is not easy to quantify and could be subjective. However, an important correlation can be made: *development complexity can be approximated to the process of identifying information about the memory usage behaviour of an application.* For example, in an explicit, fine grain model, this information is an expression of the lifetime of each object as specified by the *malloc()* and *free()* operations. Other models, such as the Metronome collector discussed in Section 2.1.6, allow the expression of other information such as the allocation rate and the λ factor.

Four Types of Information for Memory Models

Information about memory usage in an application can be expressed either for a specific application or a more general application class. The difference between general and application-specific information is mainly one of tuning in the adopted strategy; general information is used to define a memory model's basic strategy whereas application-specific information is used as a parameter to refine this strategy. For example, generational garbage collectors [58] define a basic global strategy based on the lifetime of objects but may also provide variable generational parameters that can be specialised based on information about the lifetime patterns of objects for a particular application. In some cases there is no default for this type of information. For instance, real-time garbage collectors require user-defined parameters such as the maximum allocation rate. In this case, this information is by definition application-specific rather than general. General and application-specific information can be further decomposed into local and global information, the main difference being in the granularity at which the information is specified. At one extreme, fine grain allocation and deallocation using *malloc()* and *free()* operators is local information. At the other extreme, the information in the parameters described for real-time collectors are global to the whole program. Coarse grain models such as memory pools lie between these two extremes with aggregates being specified to define the lifetime of objects. There are therefore four ways in which information relating to the memory usage of an application can be expressed:

1. Global and general: This involves using algorithms that attempt to organise allocation and deallocation strategies based on the general behaviour of a given class of programs. Two examples are generational garbage collectors [58] and memory allocators which provide variable block sizes [87] based on information of the frequency of allocation requests for a range of objects sizes.

2. Global and application-specific: This involves the developer providing available information about memory use at a global level. An example of this is the user-defined parameters such as the known maximum allocation rate specified for real-time garbage collectors. In most cases, this information is obtained through an automatic program analysis.

3. Local and application-specific: In an explicit memory model, this typically involves the developer or an analysis tool specifying *malloc()* and *free()*-like statements for individual objects or memory blocks. In automatic models this is achieved by providing special reference semantics for some objects, reference types in Java being one example.

4. Local and General: This type of information is conceptually similar to global and general information but at a finer granularity.

In order to justify a particular choice of expressing information of memory usage, a quantifying relationship would be desirable. The case for application-specific information being preferred over general information is clear when the worst case has to be considered; an application that does not fit a general model is often easy to develop and such an application will perform poorly. Nevertheless, the majority of the research into memory management has gone into identifying optimisations for the common case of the "average" application. For example, in Section 2.1.1, the term "strategies" was shown to have been coined by Neely and Johnstone [95, 77] to describe how minor modifications to policy such as when coalescing occurs and choosing the block size could be used to significantly improve the average performance of a DMA. These strategies are often defaulted to values suitable for general programs and can then be specialised through global application-specific information. However, this specialisation is rarely carried out. The case for local as opposed to global information is more difficult to argue than that for application-specific versus global information as this implies a significant development burden. It would also be beneficial to be able to quantify the relationship between the same type of information. For example, it would be useful to be able to state that the accuracy of one example of global and application-specific information (for example the maximum allocation rate) is more important than another (for example the average object size). This would allows developers to identify the strategies required to satisfy the space and time restrictions of their application. Although no attempt is made in this thesis to identify all possible ways of expressing information, a quick analysis can pinpoint the better options that address the major disadvantages of the fine grain explicit and automatic memory models identified in the previous chapter. The merits of coarse grain memory models can be argued by the development complexity of capturing the information required by these models and the space and time overheads achievable using this information. Before investigating this further, Section 3.2 provides a classification of memory models by identifying how different memory models perform in relation to the memory usage information they operate on.

Collecting Memory Usage Information

Approximating development cost to the burden of expressing information about memory-related behaviour blurs the distinction between man-hour costs incurred through manually expressing this information and the time and monetary cost of tools that discover this information automatically. In most cases, automatic techniques are used for gathering application-specific information at compile-time.[2] This is favoured by the developer as the development cost (by the definition given above) is offloaded from a man-hour cost into the cost of the tools. Tools for automatically calculating global parameters for fine-tuning garbage collectors are also common [91, 108]. The most common automated way of obtaining local and application-specific information is through escape analysis [92,

[2]The optimisations derived are sometimes called "profile-based optimisations" [18].

27, 48]. This is typically used to supplement automatic fine grain models by reducing the heap size and therefore the time for a collection. Automated approaches for collecting information could be equally useful in developing real-time applications as in non real-time ones. As a pre-deployment investment, they can provide benefits insofar as they output conservative results that encompass the worst case. It is noted in passing that in some cases, the available information can be used to derive other information. For example, the maximum allocation rate of an application – a global piece of information – can be derived using analysis of the locations of explicit allocation and deallocation calls – a local piece of information.

Another consideration is that some information of memory usage may only become available at runtime. For example, it may be the case that although the exact lifetime of an object may not be known at compile time, this information becomes known at the time the object is allocated at runtime. In this particular example, it is known that this information is of little benefit to an allocation strategy as it can be shown that any allocation strategy that attempts to use this information is guaranteed to perform almost as bad as an allocation strategy where this information is unknown [80]. However, this result can not be generalised to all types of information. For the purposes of this thesis, only information that is of benefit to a coarse grain model and can be identified at compile time is considered.

3.2 Memory management as an *Entropy Problem*

A description of the interrelation between the parameters of the evaluation metric can be argued by the *Entropy Hypothesis*. The Entropy Hypothesis states that the relationships between space, time and object information can be characterised as a form of entropy. Entropy is a term borrowed from the concept of entropy in Information Systems Theory [109] to which a parallel can be drawn. In information systems, entropy is a measure that is used to calculate the amount of information in a source or, equivalently, the *redundancy* in that source. The entropy gives a measure of the actual information in a system and dictates the maximum degree to which that system can be compressed and, thereby, the number of bits required to transmit that source. Every information source has a *maximum entropy* that sets a lower bound for the compression of that source through lossless algorithms. When a system is said to be at maximum entropy, it is implied that it exhibits maximum randomness or, equivalently, no information is known about the information source and lossless compression is impossible. This results in a large number of bits required to transmit this source. However, if certain information is known about the source, then this can be used by a compression algorithm to reduce the number of bits required for transmission. In this case, the original source is not completely random and this information can be taken advantage of during encoding. For example, fax machines take advantage of the high redundancy caused by whitespace

to compress data sent over a telephone. However, if a fax is sent in which the order of white and black dots is random, then no compression is achieved as the lack of redundancy means the source is already at maximum entropy.[3]

The analogy of memory management to entropy is best explained by means of an example. Consider an application A for which no information is available as to its memory use. Clearly, the worst case memory requirements for A are infinitely large. If A is known to make n memory requests for a maximum of k bytes each, then it is known that at most $n \times k$ bytes are required to safely execute A. If it is then known that in the worst case the difference between the number of allocation and deallocation requests carried out is t where $t \leq n$, then as little as $t \times k$ bytes are required if fragmentation is ignored. When considering fragmentation however, the memory requirements are still bounded from above by $n \times k$. Program behaviour can also be used to reduce fragmentation for infinite running programs, thereby containing the impact on both space and time requirements as described in Section 2.1.1. For example, if it is known that allocations and deallocations are done in a LIFO manner with a maximum queue size of o outstanding deallocations, then no fragmentation will occur. Memory size is therefore bounded by the maximum size of any o consecutive allocations. Clearly as more information about program behaviour becomes available, this information can be used in order to reduce the known worst case memory requirements of that program. A similar argument can be made for the temporal dimension. For example, generational garbage collectors [58] take advantage of common patterns of object lifetime to focus collection on those objects most likely to have become garbage. Similar to the concept of minimum entropy in information systems theory, an application will incur the least possible space and time overheads when:[4]

- at the point of execution when most memory is live, no garbage exits,
- no allocation or deallocation occurs during time-critical execution and
- no fragmentation ever occurs.

This is tantamount to there being complete information of the application's memory usage available, which is leveraged in the implementation of the adopted memory model. Most models can in practice achieve very small space or time overheads but can rarely minimise both. For example, a fine grain model can minimise space overheads by defragmenting at every deallocation but then the time overheads are high. At the other extreme, near-zero time overheads can also be achieved at high space costs by never deallocating objects.[5]

[3] In fact, the resulting transmission could require more bits to be sent than had no compression been attempted.

[4] Unless otherwise stated, this thesis henceforth assumes the worst case requirements rather than the average requirements during the application's lifetime are under discussion.

[5] This is never exactly zero due to the time required to update the pointer to free memory.

A Classification of Memory Models Using the Entropy Hypothesis

The following analysis of memory management techniques according to entropy is based on the hypothesis that the amount of available information about an application defines the space and time overheads of the application. As illustrated through the analogical example above, just as known information of patterns in an information source can be used to reduce the cost of transmission, so information about memory usage can be used to reduce the space and time overheads of memory management. Also, using the notion of maximum entropy, maximum randomness in an information source that results in high transmission costs can be compared to high space and time requirements in application execution. For a given amount of information about an application, a solution is defined in a memory/time trade-off space for which a function that depends on the chosen memory model will define the tradeoff in the time and space dimensions. Therefore, with the same amount of information, a tradeoff between the space and time overheads of targeting a memory model could be achieved as discussed in Section 2.1.7. The tradeoff between these dimensions is a potentially unique signature for that model and can identify at a fine granularity a ranking of models for a particular application class.[6] However, the bounds of this space are defined by the entropy of that information: with a given amount of information, there is a bound on the solution trade-off space. The Entropy Hypothesis is depicted in Figure 3.1 for a hypothetical application and memory model.[7] The available information about the application defines the space of memory/time trade-offs which memory models can achieve. If less information is available, the space and time requirements could increase but are always bounded from below by the space for which maximum information is available.

Using the Entropy Hypothesis, the investigation into the overheads of fine grain memory models described in detail in the previous chapter and the well-known overheads of the static approach, a spectrum of memory management technologies that describe the resultant overhead of some of these models can be defined. This spectrum is shown in Figure 3.2 where the space and time overheads of each solution are described in relation to the amount of information that is expressible and used by the memory model. This is based on a representative hypothetical application of non-trivial complexity and may vary for other applications. The shape of the space is likely to be not as well defined in practice as the figure would lead one to believe, but for illustrative purposes the tradeoff in space and time is captured here by the triangular shape. Another simplifying assumption made here is that the general application under investigation terminates. A case for the space occupied by coarse grain models as depicted here is made in Section 3.3.

It is noted in passing that the argument made in Section 2.1.3 that explicit memory management

[6] The assumption here is that the tradeoff is defined for the worst case values of the respective dimensions. There is a significant complexity in defining this function as discussed in Section 2.1.7.

[7] Note that the oval shape is not necessarily indicative of the true shape of this space.

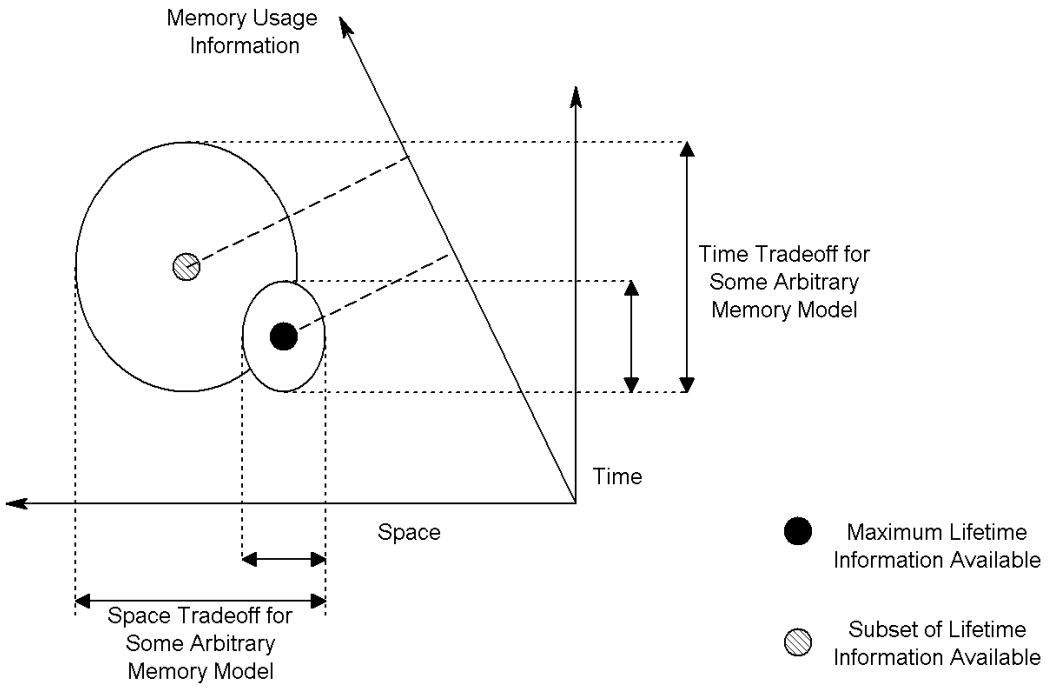


Figure 3.1: Space, Time and Memory Usage in the Entropy Hypothesis

can always be made to perform at least as well as automatic memory management is captured by the Entropy Hypothesis; the exact lifetime of objects is unknown before runtime and therefore the space-time tradeoff must be worse. The arguments made in [28, 72] where garbage collectors are argued to perform better than explicit allocation and deallocation are then captured by the tradeoff function. For example, in Figure 3.2, the points A vs. B_1 and B_2 represent the space and time costs for an application using an explicit memory model vs. an automatic memory model respectively. Clearly, the automatic memory model is more time efficient but less space efficient, at B_1. However, to achieve the same space overheads as A at B_2 the additional cost of tracing guarantees that a higher time overhead is incurred than in the explicit model.

Targeting a Memory Model

A caveat of the Entropy Hypothesis is that the decision of which memory model to use must be made before implementation of the application begins. It is commonly argued that specialised systems can not be developed in the same way as regular applications [112]. Hard real-time systems fall into this class of systems as real-time requirements demand different programming approaches. For example, conformance to the SPARK Ada subset [17] requires an application to avoid dynamic allocation. A similar argument also holds for developing software that targets a memory model. For example, if an explicit, coarse grain model such as a memory pool model is to be used, then the developer must

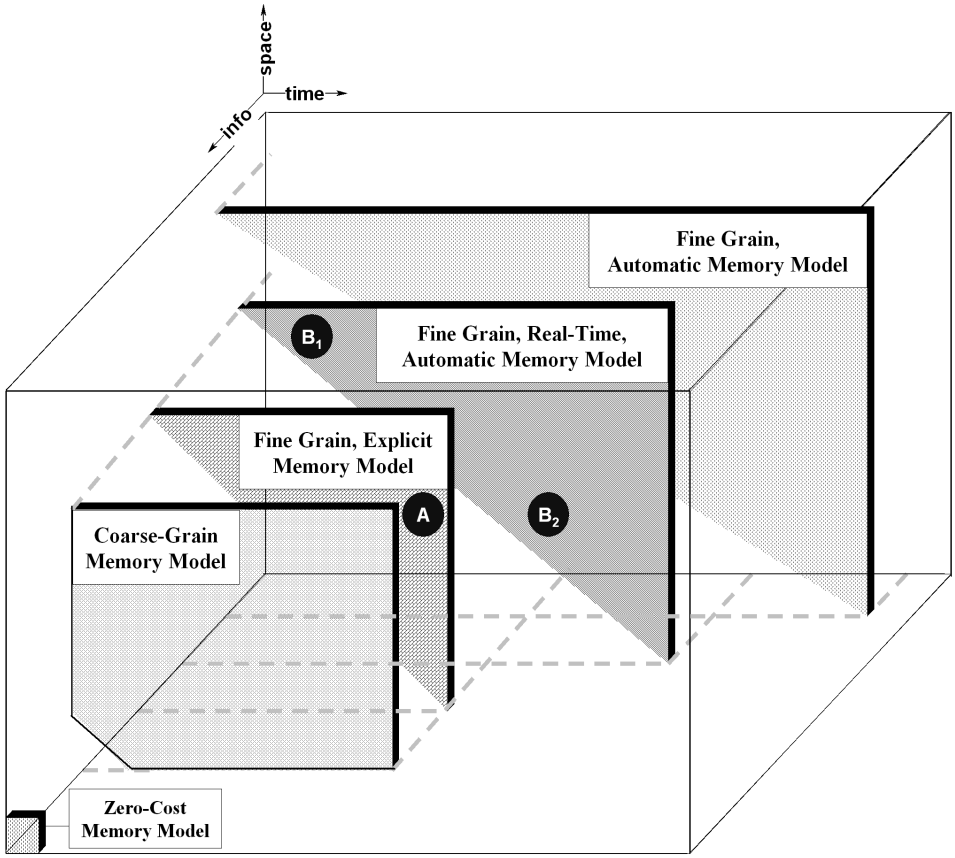


Figure 3.2: Classifying Memory Models by the Entropy Hypothesis

identify appropriate clusterings to place inside each pool. In time-critical applications, consideration must also be given to the timing requirements of tasks. In quantifying the development costs of a chosen model and the corresponding expected time and space overheads, an important assumption is made: *the developer must target that memory model.*

Targeting a memory model plays an important part in leveraging the advantages of that model. For example, if a memory-pool model is provided, then programming with a fine grain approach by placing one object in each pool will add complexity in development and maintenance and fail to achieve the reduced time overheads of the coarse grain approach. Therefore, choosing the right memory model that allows the implicit information of memory usage present in the application to be used is crucial.[8] However, there is probably no memory model suitable for all types of applications as the patterns of memory usage can not be generalised. Therefore, choosing a coarse grain approach is only of benefit if an aggregation of object lifetime is implicit in the application. If it is, then a

[8]It is noted in passing that the abstraction problem is one also faced by real-time developers in the area of scheduling. The traditional cost/deadline model fails to capture more complex abstractions of timing requirements with the result that mapping this requirements to the traditional model is often complex and inefficient.

memory model that describes a coarse grain abstraction can capture this information so that it may be used to drive the allocation and deallocation strategy. If this aggregation does not exist, then a coarse grain model is a bad abstraction to adopt and only increases development and maintenance complexity.

The first part of the thesis hypothesis states that this thesis contends that a coarse grain approach can *help* meet the challenge of memory management in real-time and embedded systems. This acknowledges the fact that a memory model that captures all information on memory usage in an application by using a single abstraction is probably unattainable. However, the fine grain approaches described in Section 2.1 clearly fail to capture enough information to achieve sufficiently low space and time requirements for constrained real-time and embedded systems. Although the Region Partitioning Model (RPM) developed in Chapter 4 allows lifetime aggregation to be captured as an alternative to that described in fine grain models, the advantages of this approach only hold if the implicit information in the program can be captured by this abstraction. Therefore, the RPM fulfills the requirements of the first part of the thesis hypothesis by providing an abstraction that captures this information. However, using the RPM to capture information outside this abstraction will only add to the development and maintenance costs and also fails to take advantage of other information that would have been better captured by an alternative abstraction.

Redefining the Term "Memory Model" to Motivate New Research

The Entropy Hypothesis defines a more abstract view of the memory management problem. This allows a comparison not only between similar models that provide little more than minor shifts between space and time requirements but also allows a comparison between intrinsically different memory models. It therefore provides a platform to compare models as different as real-time collection and the RTSJ scoped memory model. The Entropy Hypothesis also motivates a more abstract definition of what a memory model is and the functions it performs. *A memory model is defined as a mechanism that allows the expression of information of memory usage and takes advantage of this information in order to reduce time and space overheads.* The goals of a memory model are therefore twofold: providing a mechanism that allows this information to be expressed and taking advantage of this information. In promoting any memory model, the importance of making this information as easy to obtain and express as possible can not be overstated. This is the central idea of "targeting" a memory model as described above and is particularly true of real-time environments as the guarantees the memory model can provide are directly related to the accuracy and precision of this information. *The success of bringing dynamic memory models to constrained real-time environments therefore lies first in providing the right abstractions to capture information about memory usage in the application.* The ability to take advantage of this information is equally important and can be used to give a comparative assessment of different implementations of the same model.

The Entropy Hypothesis hints at the direction future research must take in order to fill the gap between the static approach and traditional fine grain models. In the absence of a proven lower-bound for the space-time tradeoff of existing memory models, there are two possible research directions that can be taken to improve on the static approach: invest further in refining existing fine grain models or derive alternative models that target regions of the trade-off space that fall between the static and dynamic approaches. In either case, the Entropy Hypothesis makes a case for memory models to allow more information to be expressed in order to reduce space and time overheads. Four questions that research must therefore address are:

1. What types of information can be captured?
2. Which single or combination of these types of information best allows the shift towards the hypothetical zero cost model?
3. What is the best way to capture this information so as to place the least possible burden on the developer?[9]
4. What is the best way to make use of the different types of information that can be captured?

This thesis considers object lifetime aggregation based on lifetime locality within program flow as one element of the set of all possible answers to the first of these questions. The motivation for a coarse grain model as a candidate solution for the second question was briefly introduced in Section 2.1.7 and is discussed in more detail in Section 3.3. The RPM developed in Chapter 4 gives a solution to the third question by providing a simple way of capturing the necessary information required by this model. The fourth question is answered in relation to the coarse grain model in Chapter 4 and the effect of using a scoped ordering[10] based on this same information is investigated in Chapter 5.

3.3 The Case for Coarse Grain Memory Models by the Entropy Hypothesis

In Figure 3.2, the space and time overheads that can be achieved by a coarse grain model are shown to improve on those exhibited by fine grain models. This section makes the case in support of this statement by arguing that a coarse grain memory model counters the two greatest sources of overheads identified in existing fine grain models in Section 2.1: fragmentation and tracing. Furthermore, the specification of the lifetime of regions by the developer at a coarse granularity becomes feasible where this is increasingly less practical in fine grain models.[11] It therefore becomes

[9] The advantages of using automatic techniques have already been briefly discussed at the end of Section 3.1.
[10] Note that the terms "scope ordering" and "stack ordering" are used interchangeably throughout this thesis.
[11] This is observed in the drive away from explicit memory models to automatic ones.

possible to leverage the benefits of existing work on offline allocation strategies as well as to consider new strategies such as the scoped ordering proposed in the RTSJ.

Fragmentation and Tracing as the Greatest Sources of Overheads

Whereas it could be argued that the analysis described in Section 2.1 for fine grain models fulfill real-time predictability requirements, the overheads of the approaches described therein may be prohibitive in resource-constrained environments. From this investigation, it is immediately apparent that the most urgent information required is that which addresses fragmentation and the cost of tracing. The Metronome collector may fail in the former case only because most applications have fluctuating characteristics such as allocation and fragmentation rates that are not well captured as global information. It fails in the latter case because it does not allow the user to express where objects are known to have become garbage. Therefore, the solution to the memory management problem for resource-constrained real-time systems could lie in an explicit model (thereby eliminating the need for tracing) and directing research at a more local characterisation of the application's information, particularly fragmentation. The Entropy Hypothesis argues the case for more information to be expressible and for this information to then be used by that model. The key problem is identifying what this information is and how it can be captured. Rather than arguing for similar global, application-specific information to be used in these models, a case for more localised information can be made. For example, global parameters could be made more localised in Metronome by being sensitive to the program's flow. Therefore, rather than there being just one integer λ factor, a number of values could be assigned that depend on the current execution trace. This could then dynamically alter the collection rate or where defragmentation is triggered. These values therefore become points on the flow graph of the application and new analysis would be required to identify how the transition between these points changes the behaviour of the collector. The feasibility of such a characterisation is unclear, both in terms of identifying this function as well as how this fits into the scheduling model. Until such research is carried out, an alternative is available in the form of coarse grain memory models.

Explicit Coarse Grain Models as a Solution to Fragmentation and Tracing

The cause of fragmentation in DMAs that adopt online allocation policies is the irregular arrival pattern of allocation and deallocation requests [78]. As described in Section 2.1, the assumption of a random arrival pattern results in worst case space requirements that are often too large for constrained real-time and embedded systems [103, 104, 105]. However, Johnstone's research [77, 78] shows that regularities in arrival patterns are often implicit in most applications. Coarse grain models provide a mechanism to capture this phenomenon in such a way that deterministic guarantees can be provided on the space and time requirements of the application while taking advantage of reduced

overheads demonstrated in the observed case. In particular, coarse grain models take advantage of the phenomenon that it is often possible to aggregate objects according to their lifetime based on some boundary criteria. For example, objects that are created close together typically have similar lifetimes due to spatial locality of reference [23]. This phenomenon is used implicitly by generational garbage collectors [58] as a general and global piece of information. For a real-time environment, a more application-specific and local approach is required as accuracy and precision is necessary in order to provide the required guarantees with low pessimism.

In the previous example, the aggregation of object lifetimes is identified by when objects are created. Consider a second example where a fine grain, first-fit policy is used in a non-terminating application that creates objects of sizes 1 to 1024 bytes. By Robson's analysis, this would require ten times the amount of live memory in the worst case. However, if it were known that any object of size 1 byte is deallocated at the same time as the object allocated immediately before that single byte object was allocated, then only nine times the amount of live memory is required. Although this information is implicit in the application and will result in lower observed space overheads in the first-fit model than would be derived from a worst case analysis, the inability to capture this information in this analysis results in pessimistic space requirements. However, a coarse grain model provides a way to express this information explicitly, thereby guaranteeing that these lower overheads are observed at runtime.

The benefits of a coarse grain approach are leveraged when the aggregation of object lifetime can be captured without additional development complexity. Therefore, since coarse grain models involve first specifying the boundaries of aggregates, the identification of these boundaries must not be prohibitively complicated. For example, if a memory pool is used whereby every object is explicitly placed in a pool based only on knowledge of the exact lifetime of that object and without any discernable underlying pattern, then targeting this model provides little advantage in terms of development complexity over using an offline algorithm that requires the exact lifetime of individual objects to be specified.[12] The developer in this case is still required to know the exact lifetime of every object and placing the object in a pool is just an expression of this information. This is an example of an application targeting the wrong memory model as discussed in Section 3.2.

Coarse grain models are not by definition explicit memory models as the deallocation point of regions could also be identified by automatic techniques. However, the reduced number of elements that need to be contended with in comparison to a fine grain model means that an explicit approach to specifying the lifetime of regions is a feasible option. In this case, tracing is therefore no longer required to identify objects that can be deallocated, thereby elimination the overheads of this operation. However, an explicit memory model has repercussions on the safety of an application in

[12]Note that only memory requirements are considered here. In practice, the choice of a memory pool model could also be based on cache considerations.

the event that the lifetime of an object specified at compile-time does not match that of the object at runtime. The garbage collection approach is to guarantee that this problem never occurs. The RTSJ's approach is to employ the scoped reference rules as described in Section 2.2 to ensure no reference can be created that could later lead to a dangling pointer. An alternative solution is to eliminate the runtime overheads of these two approaches and provide a failure mechanism in the event that an object is prematurely deallocated when the information specified on the lifetime of that object is discovered to be incorrect at runtime. A mechanism that achieves this in the RTSJ is described in Section 6.1.2.

Using the Relative Lifetime of Regions to Reduce Overheads

Information available at compile-time on object lifetimes is rarely taken advantage of in explicit, fine grain memory models. Indeed, most compilers go through great lengths to optimise for performance but few optimise for space [67, 59]. The results of research into polynomial offline algorithms are rarely carried over to real-world compile-time optimisation, firstly because, as noted by Johnstone, such optimisation is rarely necessary and secondly because specifying the lifetime of every object allocated in the application is complex in practice[13] and undecidable in general by analysis techniques [57]. However, the results of this research could prove beneficial to real-time applications if capturing this information is made more practical. A coarse grain memory model can leverage these benefits as specifying the lifetime of coarse grain entities incurs a burden on the developer that is significantly smaller than in a fine grain model. This reverses the current trend away from explicit models and towards automatic ones that incur high time overheads.

The Coarse Grain Models RPM − Unscoped, RPM − Scoped and the RTSJ's Memory Model

In order to leverage the benefits of a coarse grain model described above, a mechanism that allows aggregate boundaries and their lifetimes to be specified is required. This information is expressed in this thesis using a memory usage description language called the Region Partitioning Model (RPM) that is introduced in Chapter 4. This information can then be applied to a particular allocation and deallocation strategy in order to derive the space and time requirements of applications that target that model. In Section 4.4 and Chapter 5, two memory models are introduced: RPM − Unscoped and RPM − Scoped respectively. RPM − Unscoped describes a coarse grain model in which the strategy for region allocation can take advantage of positive results in existing research into DMAs that use offline and online algorithms. RPM − Scoped describes a coarse grain model in which the order of region allocation and deallocation follows a scoped order similar to that in the RTSJ.

[13] Indeed, the popularity of automatic memory models is that the exact lifetime of an object does not need to be specified.

In addition to providing a mechanism for targeting coarse grain models, the RPM is also used to understand the implications of the scoped ordering of regions in RPM − Scoped. As RPM − Scoped makes use of the lifetime information of regions to derive an allocation strategy, the Entropy Hypothesis would indicate that RPM − Scoped should therefore be able to provide a reduction in at least one of space and time overheads in comparison to a DMA policy that assumes a random arrival pattern of allocation and deallocation requests. Critics of the RSTJ claim that this is not the case [5]. Indeed, the execution cost of allocation and deallocation of regions is similar in the worst case to that of a coarse grain model that does not take advantage of region ordering. Moreover, the observed worst case space overheads are claimed to be infinite for some infinitely running programs and this will be proven to be the case for RPM − Scoped. Understanding the cause of this discrepancy is important in motivating an extension to address this problem.

The assumption at this stage is that RPM − Scoped and the RTSJ's scoped memory model are identical. Identifying the differences between the two models highlights two additional problems particular to the RTSJ approach: the first is that the RTSJ makes it unnecessarily complex for applications to target its scoped memory model. The second problem is that the RTSJ does not require key implementation functionality in the virtual machine that can take full advantage of expressed information that can minimise space and time overheads.

3.4 Defining a Memory Model

The definition of a memory model on page 62 in Section 3.2 describes two functions: expression of known memory usage behaviour and taking advantage of that information. Consequently, the process of expressing memory usage can be separated from the mechanics of physical allocation and deallocation performed by the runtime and this makes it possible to evaluate the expressiveness of a model independently of its implementations. This separation is compatible with Johnstone's and Neely's separation of policy and mechanism [95, 77] with these two components forming part of what is described here as the memory subsystem. Consider the following three examples of expressions of known memory usage information:

- **Example 1:** In the fine grain memory model, the lifetime of a data structure starts when a *malloc()* call is made and ends either when a *free()* call is made or when that structure can no longer be accessed.

- **Example 2:** In a collection data structure, the object representing the collection is known to always live longer than the items which it indexes. Furthermore, the semantics of the program show that items are created and removed in a queue-like fashion in groups of a known number of objects.

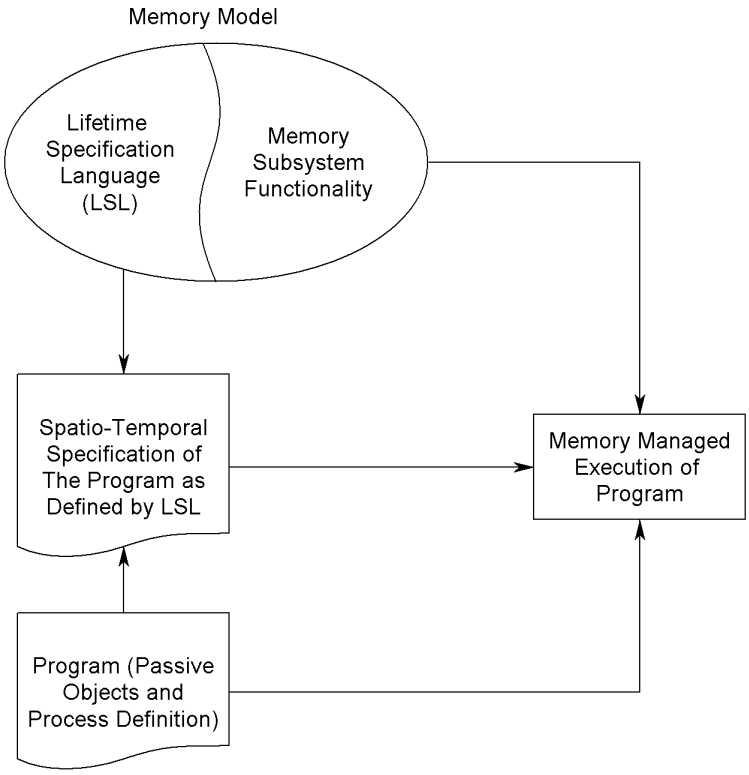


Figure 3.3: Separating the Functionality of Memory Models

- **Example 3:** In a program without global variable access, a method that takes no parameters and returns no parameters is always guaranteed to be able to free any memory allocated within that method.

The three examples are natural language descriptions of known object lifetime that can be used by a memory model. The ambiguity of natural language makes in virtually impossible to translate information in this form to anything a memory model could make use of and therefore a language to describe known lifetime behaviour is required. This language would be unique to a memory model in that it provides the appropriate descriptors to target that model. This language is called the *Lifetime Specification Language* of the memory model and the information described in this language for a particular application is the *Spatio-Temporal Specification* of that application. The examples highlight how there can be several criteria that can be used to specify temporal information. Examples 1 and 3 are based on global information about the program whereas Example 2 is based on local information. Different dynamic memory models may or may not take advantage of the information in these statements to organise allocation and deallocation policies accordingly. If this information can not be expressed, then the memory model will not be able to use this information to optimise these policies. If this lifetime information can be expressed in a memory model, then a

good implementation of that model will take advantage of this information. The notion of targeting a memory model can be described in this context as deciding what information is best expressed in that model in order that an implementation can take advantage of it.

A memory model can therefore be further defined in concert with the evaluation metric as a composition of two components as shown in Figure 3.3. The first is a *Lifetime Specification Language* (LSL) that describes how memory usage information such as object lifetime information is to be specified by the developer in a *Spatio-Temporal Specification* (STS). The second component is the *memory subsystem* that translates this information into an appropriate memory allocation and deallocation strategy. The evaluation metric for memory models can be applied to the LSL and memory subsystem independently. This means that the space and time overheads incurred through the LSL and memory subsystem can be evaluated independently based on the memory usage information available to each one. The LSL and memory subsystem specifications can be applied to a program in order to derive the memory requirements of the system, the former in the way it allows memory usage to be expressed and the latter in the way memory is physically used. Some temporal requirements can be expressed in the STS but as these are eventually realised in the memory subsystem, the temporal costs of the memory model are only defined in the memory subsystem. A formalisation of the definition of a memory model follows.

A memory model $\mathcal{M}$ is defined as a tuple $\mathcal{M} = \langle \mathcal{M}^{\mathrm{LSL}}, \mathcal{M}_{\mathrm{MS}} \rangle$ where LSL denotes the Lifetime Specification Language of $\mathcal{M}$ and MS denotes the memory subsystem of $\mathcal{M}$. The shorthand notation $\mathcal{M}_{\mathrm{MS}}^{\mathrm{LSL}}$ can be used when describing this memory model. When $\mathcal{M}$ is used with an application P, the memory requirements are described as $\mathrm{MemReq}^{\mathcal{M}}(P)$. $\mathrm{MemReq}^{\mathcal{M}}(P)$ is a function of $\mathrm{MemReq}^{\mathcal{M}^{\mathrm{LSL}}}(P)$ and $\mathrm{MemReq}^{\mathcal{M}_{\mathrm{MS}}}(P)$. In general this function is simply an addition but this is not always the case as described for the Metronome collector below. $\mathrm{MemReq}^{\mathcal{M}^{\mathrm{LSL}}}(P)$ denotes the memory requirements assuming a theoretical memory subsystem that is biased completely towards reducing space as opposed to time overheads. $\mathrm{MemReq}^{\mathcal{M}_{\mathrm{MS}}}(P)$ denotes the additional memory overheads that are incurred in the memory subsystem, in particular the mechanisms costs as well as the cost of the adopted allocation and deallocation policy (including fragmentation). The time overheads are denoted as $\mathrm{TimeReq}^{\mathcal{M}_{\mathrm{MS}}}(P)$ which is a model-specific description of the overheads of dynamic memory allocation and deallocation, defragmentation and any required mechanism for identifying garbage. The cause of these overheads would have to be stated explicitly in order that it is understood where each overhead applies in the programs execution. Following are examples of memory models described using this notation.

Simple Garbage Collectors

In a garbage collector memory model $\mathcal{M}$ which uses the Traditional Garbage Collector LSL (called TGC), TGC requires that only the minimum amount of memory needed by the application be

provided as a parameter. For an application P, this is the maximum size of all live objects at any point during program execution and is equal to $\text{MemReq}^{\mathcal{M}^{\text{TGC}}}(P)$. The operation of the DMA in the garbage collector together with the mechanism used to find garbage collectively form the memory subsystem and define $\text{MemReq}^{\mathcal{M}^{\text{TGC}}_{\text{DMA}}}(P)$.

The Obsessive Garbage Collector (OGC) is a memory model with LSL TGC and a memory subsystem OGC. OGC is defined such that a collection and compaction cycle is carried out every time there is the possibility that garbage might exist in the system. A mark-sweep garbage collector which marks, sweeps and compacts with every dereference is an example of an OGC, as is a reference-counting collector which detects cycles and deletes and compacts every time a reference count drops to zero. For an arbitrary program P that uses the OGC, $\text{MemReq}^{\mathcal{M}^{\text{TGC}}_{\text{OGC}}}(P)$ is approximately equal to $\text{MemReq}^{\mathcal{M}^{\text{TGC}}}(P)$ as $\text{MemReq}^{\mathcal{M}_{\text{OGC}}}(P)$ only includes the cost of maintaining a pointer to the first free location and other minor machine-specific overheads. Assuming a work-based model, $\text{TimeReq}^{\mathcal{M}_{\text{OGC}}}(P)$ would describe a large value for every pointer dereference that is equal to the worst case time for finding garbage, freeing this memory and then carrying out compaction. At allocation points however, $\text{TimeReq}^{\mathcal{M}}(P)$ would describe a small value as allocation involves only updating the head pointer and returning a pointer to the allocated memory to the application.

The Lazy Garbage Collector (LGC) also uses TGC for a LSL but, in contrast to the OGC, uses a memory subsystem called LGC that never scans for garbage or deallocates memory. $\text{MemReq}^{\mathcal{M}^{\text{TGC}}_{OGC}}(P)$ lies between the specified $\text{MemReq}^{\mathcal{M}^{\text{TGC}}}(P)$ and ∞ for infinitely running programs with at least one allocation site in an infinite loop as $\text{MemReq}^{\mathcal{M}_{\text{LGC}}}(P)$ would be of size ∞ in this case. However, $\text{TimeReq}^{\mathcal{M}_{LGC}}(P)$ would give a value of zero as the time overhead at pointer dereference and still maintain a small value at allocation points.

OGC and LGC effectively mark the two edges of the space-time tradeoff for garbage collectors as they share the same LSL but have memory subsystems that maximise opposite sides of the tradeoff. Conventional garbage collectors use techniques in the memory subsystem such as carrying out defragmentation only at specific points in order that a better balance is prescribed in this tradeoff.

The Metronome Collector

The Metronome garbage collector [12, 14, 13] (Metronome = $\langle \mathcal{M}^{\text{Spec}}, \mathcal{M}_{\text{DMA}} \rangle$) was described in detail in Section 2.1.6. $\text{TimeReq}^{\mathcal{M}_{DMA}}(P)$ is effectively the percentage of execution time that is assigned to the collector. The collector allows five pieces of global, application-specific information

to be specified.[14] In this case, $\text{MemReq}^{\mathcal{M}^{\text{Spec}}}(P)$ and $\text{MemReq}^{\mathcal{M}_{\text{DMA}}}(P)$ are defined in a recursive relationship. Given the required mutator utilisation and the allocation rate and garbage collection rate, $\text{MemReq}^{\mathcal{M}^{\text{Spec}}}(P)$ is calculated by ignoring fragmentation. In calculating $\text{MemReq}^{\mathcal{M}_{\text{DMA}}}(P)$, fragmentation is then considered and the extra space required added to $\text{MemReq}^{\mathcal{M}^{\text{Spec}}}(P)$ of the first iteration. In order to maintain the required mutator utilisation, the execution time of the collector may have to be decreased in order to take into consideration the execution time of the defragmentation algorithm. This therefore decreases the rate of collection and further increases the value of $\text{MemReq}^{\mathcal{M}^{\text{Spec}}}(P)$.

Coarse Grain Memory Models with Known Region Lifetimes

In a coarse grain memory model, the lifetime information is derived from information of the semantics of the program and must be made explicit in the STS. The memory subsystem in its simplest form would require the DMA to place objects in the regions specified in the LSL as well as to allocate a region in time for when a memory region is marked to begin its lifetime and deallocate sometime later when the region is no longer required. The DMA allocation/deallocation and defragmentation strategy would also be defined in the memory subsystem. The LSL for a coarse grain model needs to define the way the developer expresses how objects are aggregated and how the lifetime of these aggregates is defined.

The Region Partitioning Model (RPM) developed in the next chapter is a LSL used to define object lifetime aggregation based on lifetime locality within program flows. The RPM thereby allows developers to adopt a coarse grain approach by expressing the known lifetime of objects with a minimal amount of input. The assumption here is that the application exhibits this aggregation and is therefore a suitable candidate for targeting this model. The RPM is key in proving all three parts of the thesis hypothesis. Firstly, it provides a model for expressing memory usage that abstracts memory concerns away from code and therefore provides a development platform that helps minimise the cost of targeting coarse grain models. Secondly, it allows an evaluation of the scoped, coarse grain approach that motivates appropriate extensions that address the problems of this model. Finally, it can be applied to the RTSJ to express region lifetime and is used to identify where the RTSJ fails to take full advantage of the scoped model. A number of different versions of the RPM are derived together with associated memory subsystems for two different memory models: RPM − Unscoped

[14] These are:

1. the proportion of memory devoted to the heap,
2. a lower bound on the average object size,
3. a lower bound on the number of pointers per word,
4. an upper bound on the fraction of non-`null` references and
5. a fragmentation factor λ.

and RPM − Scoped. The values of $\text{MemReq}^{\mathcal{M}^{\text{LSL}}}(P)$, $\text{MemReq}^{\mathcal{M}_{\text{MS}}}(P)$ and $\text{TimeReq}^{\mathcal{M}_{\text{MS}}}(P)$ are derived in each case. The model is grown from a simple sequential description of executing aggregates to more complex descriptions with conditional and iterative control flows. Sequential flows in multi-threaded programs are also considered with the ability for regions to share objects between threads and for threads to synchronise with each other.

3.5 Summary

The Entropy Hypothesis introduced in this chapter provides a basis for understanding the differences between memory models. It contends that the three dimensions of development cost, space overheads and time overheads are closely related. The challenge is to find a memory model that provides the right balance across these three parameters for the required class of applications. Achieving this requires a mechanism that makes it as easy as possible to express known memory usage (both in new and existing code) and providing an implementation that can make the best use of this information. The definition of a memory model was built around these requirements with a separation being proposed between the way memory usage is expressed (the LSL) and the way this information is used (the memory subsystem).

It was argued that an explicit, coarse grain memory model would be expected to be of benefit in reducing space and time overheads as it allows a way of expressing information that directly addresses fragmentation and tracing. The assumption here is that object aggregation is implicitly present in the application, thereby making it a suitable candidate for being targeted using a coarse grain model. By the Entropy Hypothesis, taking advantage of available information on the lifetime of regions could further reduce both space and time overheads, a fact proven by research into offline algorithms. However, adopting a scoped approach based on program flow (as is done in the RTSJ) in order to take advantage of this information requires further investigation. This would allow an understanding of where the benefits of this approach lie and where its use would be counter-productive. The challenge is then to find the best way to use this memory model in a Java environment while providing alternatives when it can be shown that a scoped approach is of no benefit. If this is achieved, then the result is a memory model for the RTSJ that can take full advantage of the scoped, coarse grain approach without high development and maintenance costs.

Chapter 4

The Region Partitioning Model for Evaluating Coarse Grain Memory Models

Targeting coarse grain memory models requires an appropriate Lifetime Specification Language (LSL). The Region Partitioning Model (RPM) developed in this chapter demonstrates how such a LSL can be developed to express object aggregation by which regions boundaries are defined based on object lifetime locality within program flows. The relative lifetime of a region is also defined against that of other regions across the same flows. Developing a complete model that captures all possible information that might be expressed by an application exhibiting this aggregation is outside the scope of this thesis. Rather, the thesis hypothesis requires that it is shown that providing an environment where this aggregation is specified as a separate concern to program logic reduces development and maintenance complexity. The goal of this chapter is therefore to provide a framework for expressing this information and then to illustrate how the model can be gradually extended. This is achieved by first describing a minimal subset of a RPM full model that includes only a simple sequential description of executing aggregates and then extending this model to more complex descriptions with conditional and iterative control flows as well as concurrent flows for multi-threaded programs. Further extensions to the model would follow a similar pattern to that used to derive the different versions of the RPM described in this chapter. At every stage, a description of the model, its execution semantics and a set of safety checks[1] required to guarantee a correct Spatio-Temporal Specification (STS) are specified. For each model, an expression of the memory requirements of programs using that model must then be derived. This allows the developer to determine the memory requirements of an application before considering the additional overheads of an underlying memory subsystem.

[1]These checks enable the developer to ensure that there is no ambiguity as to the lifetime of a region, thereby guaranteeing that an offline algorithm or an online DMA can always determine the lifetime of a region.

The unique aspect of the RPM is that the allocation and deallocation points are not fixed but can be moved about as required with the constraint that memory is allocated before it is required and not deallocated too early. As the different versions of the RPM are developed to capture additional control flows and multi-threading, *constrained* versions are also developed that allow the shifting of allocation and deallocation according to some constraint. There are at least two benefits of this approach. Firstly, offline algorithms such as that developed in Chapter 5 for deriving a scoped model can be described as a transformation from an unconstrained model to a constrained one. Properties of this transformation – for example that the lifetime of regions are not made ambiguous in this transformation – can then be checked against the criteria for correctness of the constrained version of that model. Existing offline algorithms only consider deriving a fixed location in memory for allocating and deallocating memory at the programmer-defined points. A scoped memory model on the other hand shifts these points in order to provide an ordering on region allocation and deallocation without the exact location being derived.[2] The RPM provides a mechanism to investigate the impact of this approach on the memory requirements of an application as required by the second part of the thesis hypothesis. A second benefit of the ability to shift allocation and deallocation points which is noted here but not pursued further relates to real-time requirements of applications. Memory allocation and deallocation is unique in that the cost of these operations can be moved around in the program without side-effects on program logic. The caveat is that memory is allocated by the time it is required and not deallocated prematurely. Therefore, if an allocation request occurs at some point in a task that inhibits that part of the task from completing on time, then the allocation point can be moved to an earlier point in the task. The constrained versions of the RPM allow this transformation to be expressed while deriving the new memory requirements of the system and guaranteeing that memory is deallocated at some point in the future.

This chapter begins in Section 4.1 by describing a sequential version of the RPM where no inter-region conditional or iterative flows are defined. This demonstrates how information on region lifetime in these programs can be expressed and how the memory requirements are calculated. Multi-threading is also considered for programs with sequential flows in Section 4.2 where it is possible to describe the sharing of objects between threads as well as thread synchronisation. Section 4.3 considers programs with conditional and iterative flows. As the specification of region lifetime when considering both these control flows becomes significantly complex, it is shown how a simpler specification can be opted for at the expense of greater memory overheads. A summary of further extensions to the RPM that can be carried out is given in Section 8.3. As the RPM is an abstract model that does not rely on any particular underlying memory system (it is a Lifetime Specification Language), its purpose is only to allow the expression of lifetime information in a coarse grain

[2]The advantages of this approach are investigated in Section 7.2.

model. In Section 4.4 a brief description is provided of how this information can be applied to existing DMAs that use online and offline allocation strategies. A simple memory subsystem is described that, together with an STS specified in the language of the RPM, provides a complete memory model.

4.1 The RPM with Sequential Flow: $\text{RPM}^{\rightarrow}$

In $\text{RPM}^{\rightarrow}$, a single-threaded program $P = \langle R, <, \mathfrak{E} \rangle$ is made up of a set of regions $R = \{r_1, r_2, \ldots, r_n\}$ and a total ordering on R, $<$, such that if $r_i < r_j$, then $i < j$. $>$, $\geq$ and $\leq$ are similarly defined. The short notation $r_{\overrightarrow{i,j}}$ is used to denote a contiguous subsequence of R, $r_i, \ldots, r_j$. Each of the regions $r_i \varepsilon R$ creates a number of objects for which memory is required and the size of which is denoted as $\mid r_i \mid$. All memory required to execute each region must be allocated at some point *before* the start of execution of that region. For a region r_i, the latest point in the program at which this memory can be allocated is just before r_i begins to be executed. This point is called the Latest Allocation Point (LAP) of r_i and is denoted as $\text{LAP}(r_i) = r_i^-$ where r_i^- is the point in the program just before the first instruction of r_i. At some point during the execution of the program, the memory of a region r_i may be reclaimed if no objects created in r_i are required after that point. This point is specified in the Earliest Deallocation Point (EDP) of r_i and is defined by the developer in the STS of P as an element of the set $\mathfrak{E}$. The EDP of region r_i for programs in $\text{RPM}^{\rightarrow}$ is denoted as $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_i) = r_j^+$ where $j \geq i$ and r_j^+ is the point in the program just after the last instruction of r_j. No memory can be allocated or deallocated inside a region and these inter-region locations are the only places in which memory can be allocated and deallocated. Note that the location between regions r_i and r_{i+1} contains both the points r_i^+ and r_{i+1}^-. The notation r_i^+ and r_{i+1}^- as used when describing the allocation and deallocation points of a region respectively occur in the order r_i^+ followed by r_{i+1}^- . This therefore requires that the operation that deallocates memory for regions occurs before that which allocates memory for others. $<$ is extended from a total order on the regions in R to include these inter-region locations. Therefore, for all regions r_i in a program, $r_i < r_i^+ < r_{i+1}^- < r_{i+1}$.

The criteria by which the deallocation point of a region is identified is not specified at this point. However, it can be assumed for the time being that the this point depends on the object reference graph of the whole application so that no dangling pointers occur. Therefore, if some object created in r_i holds a reference to some object created in r_j, then $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_i) \geq \text{EDP}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_j)$ both if $r_i < r_j$ and if $r_j < r_i$. A discussion of how deallocation points are identified is given in Section 6.1. Program execution is defined using $<$: the first region in the order is executed first, the last region in the order is executed last, all regions are executed and if $r_i < r_j$, then r_i is executed before r_j.

Given a program $P = \langle R, <, \mathfrak{E} \rangle$ and for each region r_i in R, the EDP of r_i in $\mathfrak{E}$, and by

assuming that the memory of r_i is allocated at r_i^- (i.e. $\text{LAP}(r_i) = r_i^-$), the minimum memory requirements at each point in the program, and therefore the minimum memory requirements of the entire program, can be derived. The minimum memory requirements when executing r_i is denoted as $\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_i)$ for some memory model $\mathcal{M}^{\text{RPM}^{\rightarrow}}$ that uses $\mathcal{M}^{\text{RPM}^{\rightarrow}}$ as its LSL[3] and can be found by summing the memory required by all regions between the start of the program and r_i and which can only be deallocated at some point after r_i has finished executing:

$$\begin{aligned} \text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_i) \quad = \quad & \sum\{\mid r_x \mid \; ; \; r_x \leq r_i \wedge \\ & \text{EDP}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_x) \geq r_i^+\} \end{aligned} \tag{4.1}$$

The minimum memory requirements of executing a subsequence of P, $r_{\overrightarrow{i,j}}$, is therefore the maximum of the memory requirements when executing any region in $r_{\overrightarrow{i,j}}$. This value is denoted as $\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_{\overrightarrow{i,j}})$. Applying this expression to P gives the value for the memory requirements of the whole program, $\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(P)$, as introduced in Section 3.4:

$$\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_{\overrightarrow{i,j}}) = \max(\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_x)); \quad \forall\, r_x \varepsilon r_{\overrightarrow{i,j}} \tag{4.2}$$

$$\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(P = \langle r_{\overrightarrow{1,n}}, <, \mathfrak{E}\rangle) \quad = \quad \text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_{\overrightarrow{1,n}}) \tag{4.3}$$

The Constrained $\text{RPM}^{\rightarrow}$**,** $\text{cRPM}^{\rightarrow}$

Equation 4.1 is derived by assuming that the memory required to execute some r_i is allocated just before r_i begins executing and is deallocated exactly at $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_i)$. Consider a LSL $\text{cRPM}^{\rightarrow}$ for a memory model $\mathcal{M}$ which places constraints on the pattern of region allocation and deallocation based on the lifetime and size of these regions. The previous assumption may not hold for $\mathcal{M}^{\text{cRPM}^{\rightarrow}}$ if the constraints made by $\text{cRPM}^{\rightarrow}$ can not identify exactly the allocation and deallocation points specified in the EDPs and LAPs of each region. Therefore, memory for some region r_i may need to be allocated at the start of an earlier region than r_i and deallocated at some point after that specified in the EDP of r_i. These points are expressed in the Real Allocation Point (RAP) and Real Deallocation Point (RDP) specifications of r_i respectively and for memory model $\mathcal{M}^{\text{cRPM}^{\rightarrow}}$ are denoted as $\text{RAP}^{\mathcal{M}^{\text{cRPM}^{\rightarrow}}}(r_i)$ and $\text{RDP}^{\mathcal{M}^{\text{cRPM}^{\rightarrow}}}(r_i)$. The notation of a program in $\text{RPM}^{\rightarrow}$ is extended to allow the specification of the RAP and RDP of every region. Therefore, programs in $\text{cRPM}^{\rightarrow}$ are defined as $P = \langle R, <, \mathfrak{E}, \mathfrak{A}, \mathfrak{D}\rangle$ where $\mathfrak{A}$ is a set of RAPs that each describe the allocation point of

[3]Recall that in addition to this, the memory subsystem may choose to delay or preempt deallocation and allocation respectively. It may also be subject to other overheads such as fragmentation. The focus at this point is on the LSL of the coarse grain memory model rather than its memory subsystem.

a region and $\mathfrak{D}$ is a set of RDPs that each describe the deallocation point of a region. Note that $\mathfrak{E}$, the set describing the original deallocation point of a region is still an element of the program description in cRPM$^{\rightarrow}$. To find the memory requirements of a region in this case, the regions that contribute to the memory requirements at r_i are derived and summed. A region r_x contributes to the memory requirements at r_i if either r_x is executed before r_i and is deallocated after it or if it is executed after r_i and is preallocated before it:

$$\begin{aligned} \mathrm{MemReq}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_i) &= \sum\{\mid r_x \mid \ ; \\ &\quad (r_x \leq r_i \wedge \mathrm{RDP}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_x) \geq r_i^+) \vee \\ &\quad (r_x > r_i \wedge \mathrm{RAP}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_x) \leq r_i^-)\} \end{aligned} \tag{4.4}$$

Since by definition (this is enforced by a set of requirements developed below), $\forall\, r_x \leq r_i$ it is always true that $\mathrm{RAP}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_x) \leq r_i^-$ and $\forall\, r_x > r_i$ it is also always true that $\mathrm{RDP}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_x) \geq r_i^+$, then Equation 4.4 can be rewritten as:

$$\begin{aligned} \mathrm{MemReq}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_i) &= \sum\{\mid r_x \mid \ ; \\ &\quad (r_x \leq r_i \wedge \mathrm{RDP}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_x) \geq r_i^+ \wedge \mathrm{RAP}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_x) \leq r_i^-) \vee \\ &\quad (r_x > r_i \wedge \mathrm{RAP}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_x) \leq r_i^- \wedge \mathrm{RDP}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_x) \geq r_i^+)\} \end{aligned} \tag{4.5}$$

Using simple boolean logic, the expression on the right-hand-side can be reduced to $(a \wedge b \wedge c) \vee (\neg a \wedge b \wedge c)$ which reduces to $b \wedge c$. Therefore, Equation 4.5 becomes:

$$\begin{aligned} \mathrm{MemReq}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_i) &= \sum\{\mid r_x \mid \ ; \\ &\quad \mathrm{RDP}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_x) \geq r_i^+ \wedge \\ &\quad \mathrm{RAP}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_x) \leq r_i^-\} \end{aligned} \tag{4.6}$$

This means that a region r_x contributes to the memory requirements at r_i if r_x is allocated before r_i and is deallocated after it. The location of r_x in relation to r_i is therefore irrelevant and is left out of the equation. This is only true due to a set of rules on the specification of allocation and deallocation points that will be derived shortly. The memory requirements of a subsequence of P, $r_{\overrightarrow{i,j}}$ is therefore the maximum of the memory requirements of each r_i in this subsequence and the memory requirements of P follows:

$$\mathrm{MemReq}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_{\overrightarrow{i,j}}) = \max(\mathrm{MemReq}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_x)) \ \ ; \ \forall\, r_x \varepsilon r_{\overrightarrow{i,j}} \tag{4.7}$$

$$\text{MemReq}^{\mathcal{M}^{\text{cRPM}^{\rightarrow}}}(P = \langle r_{\overrightarrow{1,n}}, <, \mathfrak{E}, \mathfrak{A}, \mathfrak{D}\rangle) = \text{MemReq}^{\mathcal{M}^{\text{cRPM}^{\rightarrow}}}(r_{\overrightarrow{1,n}}) \tag{4.8}$$

Clearly, Equation (4.1) is a special case of Equation (4.4). The condition for adding $\mid r_x \mid$ to the memory requirements at r_i in Equation (4.6) resolves to the same condition as that in Equation (4.1) when for every r_x where $r_x \leq r_i$, $\text{RDP}^{\mathcal{M}^{\text{cRPM}^{\rightarrow}}}(r_x) = \text{EDP}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_x)$ and $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_x)$ is still greater or equal to r_i^+. Also, for all r_x where $r_x > r_i$, $\text{RAP}^{\mathcal{M}^{\text{cRPM}^{\rightarrow}}}(r_x) = r_i^-$.

It is trivial to prove that given a set of allocation and deallocation points for regions in a program P, delaying a deallocation point can increase the memory requirements of P but not decrease them. Similarly, allocating memory earlier than the provided allocation point can only increase the memory requirements of P. It therefore follows that Equation 4.3 does indeed give the minimum memory requirements of a program and this occurs when a memory model $\mathcal{M}$ can guarantee that for each region r_i in P, $\text{LAP}(r_i) = \text{RAP}^{\mathcal{M}}(r_i) = r_i^-$ and $\text{EDP}^{\mathcal{M}}(r_i) = \text{RDP}^{\mathcal{M}}(r_i)$.[4] Ensuring that this is the case is a sufficient but not necessary test to guarantee that using $\mathcal{M}$ (without considering overheads of the memory subsystem) will only require the minimum amount of memory necessary. The test is not necessary as some modification of allocation and deallocation points may increase the memory requirements at certain points in P, but not to an extent which is greater than $\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(P)$. An optimal test is to derive $\text{MemReq}^{\mathcal{M}}(P)$ for the memory model $\mathcal{M}$ and then compare this value to the unconstrained version of that memory model.

Finally, execution in $\text{cRPM}^{\rightarrow}$ is defined in the same was as for $\text{RPM}^{\rightarrow}$ by using $<$.

Correctness of $\text{RPM}^{\rightarrow}$ and $\text{cRPM}^{\rightarrow}$ Specifications

The STS of a memory model needs to be checked for errors before being used in an application or an algorithm. For most memory models such as garbage collectors, this is trivial to do and requires only a validation of the user-defined parameters. For memory models that require more input from the developer, a check on the validity of this input could reduce runtime errors that could compromise the system, particularly in hard real-time environments. For example, a memory pool model might require static checks to ensure that a particular memory pool is created before any objects that are destined for allocation in it. In the different versions of the RPM developed in this thesis, the *correctness* of an STS specification first requires a number of sanity checks such as ensuring that no two regions are given the same name. These sanity checks are assumed to be carried out and are not considered further. The STS also includes an appropriate listing of allocation and deallocation points for regions as specified by their EDP, RAP and RDP in the sets $\mathfrak{E}$, $\mathfrak{A}$ and $\mathfrak{D}$ respectively. This information must be checked in order to validate an STS. Again, a number of sanity checks need to

[4]Note that no subscript is required for LAP as it is defined in the same way for all versions of the RPM developed in this chapter.

be carried out first. This might include ensuring that every region has a specified EDP, RAP and RDP in these sets. An important check that is investigated more thoroughly in the different versions of the RPM developed in this thesis is that which ensures that a region always has the memory it requires allocated for it before it begins execution. Similarly, in order to avoid memory leaks, it must be ensured that once a region no longer needs the memory allocated for it, that memory is deallocated. These checks are carried out by ensuring that a number of criteria for the elements in $\mathfrak{E}$, $\mathfrak{A}$ and $\mathfrak{D}$ are met. For example, if a region is specified to have an allocation point that comes after that region is executed, then the specification of the RAP for that region is incorrect. This implies that the whole STS is also incorrect. Loosely speaking, for all the different versions of the RPM developed, an STS is correct if for all executions[5] of all regions r_i in the program P:

1. (a) r_i has at least one deallocation point specified in EDP(r_i)[6] in $\mathfrak{E}$,
   (b) exactly one of these deallocation points is guaranteed to be reached and
   (c) any deallocation point must be reachable only after r_i is executed.
2. If the set $\mathfrak{A}$ forms part of the STS specification (as in the constrained versions of the RPM such as cRPM$^{\rightarrow}$), then
   (a) r_i must have at least one allocation point specified in RAP(r_i)[7] in $\mathfrak{A}$,
   (b) exactly one of these allocation points is guaranteed to be reached and
   (c) any allocation point specified in RAP(r_i) must be reachable before r_i is executed (that is at r_i^- at the latest).
3. If the set $\mathfrak{D}$ forms part of the STS specification, then three checks need to be carried out:
   (a) r_i must have at least one deallocation point specified in RDP(r_i) in $\mathfrak{D}$ for every one allocation point specified in RAP(r_i),
   (b) exactly one of these deallocation points is guaranteed to be reached for every allocation point specified in RAP(r_i) and
   (c) any deallocation points specified in RDP(r_i) must come at the earliest at the first deallocation point specified in EDP(r_i) or at some point that ensures that r_i is not executed.

The constraints will become more clear when considering the correctness criteria for each model in turn. As already noted, the intuition behind these rules is to ensure that regions can not begin executing before memory is allocated for them and that memory is guaranteed to be deallocated

[5]Without going into further detail at present, it is noted that in those versions of the RPM that provide for iteration, a region may execute several times.
[6]Some versions of the RPM will have multiple deallocation points, each one specified in the EDP or RDP.
[7]Some versions of the RPM will have multiple allocation points, each one specified in the RAP.

at some point. In particular, in programs with conditionals, some regions may not execute at all making this requirement more complex to ensure. Regions in these programs could have more than one allocation and deallocation point specified for each execution of a region. In the different non-constrained versions of the RPM developed, only the first point above needs to be checked whereas all three points must be checked in the constrained versions. Note that this thesis makes no claim to a formal proof that the criteria for correctness for a memory model hold as this is outside the scope of the thesis' goals. Rather, the correctness criteria developed for each model are constructed based on the observation of how execution in these models is defined and serve as an informal basis for such a proof and a guide to the implementation of these checks.

For the RPM$^{\rightarrow}$ and cRPM$^{\rightarrow}$ LSLs, the criteria for correctness are trivial. In the case of RPM$^{\rightarrow}$ in which only the set $\mathfrak{E}$ is specified, correctness requires only ensuring that for all regions r_i, $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_i)$ is specified once, thereby satisfying Step (1a), and $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_i) \geq r_i^+$, thereby satisfying Step (1c). Since all regions are executed in RPM$^{\rightarrow}$, Step (1b) above always holds as long as $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_i)$ is only specified once. In cRPM$^{\rightarrow}$, an RAP and RDP must be specified once for every region and the invariant $\text{RAP}^{\mathcal{M}^{\text{cRPM}^{\rightarrow}}}(r_i) \leq \text{LAP}(r_i) = r_i^- < r_i < \text{EDP}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_i) \leq \text{RDP}^{\mathcal{M}^{\text{cRPM}^{\rightarrow}}}(r_i)$ must hold for all regions r_i. These two rules in effect fulfill all of Steps (2) and (3). Note that this invariant was used in deriving Equation 4.5 from Equation 4.4.

4.2 Multi-Threading in cRPM$^{\rightarrow}$

Real-time systems are by definition multi-threaded[8] and therefore the RPM needs to be extended to incorporate concurrent execution flows. There are three possible approaches that are investigated, the first involves independent thread execution where objects are not shared across threads (except in one specified special case) whereas the second allows objects created in one region to be referenced by objects in another. Sharing of data introduces a degree of indeterminacy in the lifetime of regions as these become dependent on the execution progress of other tasks.This leads to unnecessary pessimism in the worst case memory requirements of programs. To address this problem, the RPM is further extended to allow synchronisation points across threads, thereby reducing the amount of memory required. These three extensions to the RPM are investigated in turn.

4.2.1 Multi-Threading with Sharing of Static Objects: RPM$^{\parallel}$ and cRPM$^{\parallel}$

RPM$^{\rightarrow}$ can be extended to a multi-threaded model in which threads share data only through objects that are allocated statically at a pre-mission phase and can not be deallocated. Shared objects are created in a memory region called the immortal region and are accessible by all threads. Although not a strict requirement of the model, it can be assumed that updates to shared data are made

[8]The terms "thread" and "task" are used interchangeably unless specified.

through invocations on synchronised methods. A formal representation of the model together with the functions for calculating memory requirements of programs follows.

In RPM$^{\|}$, a multi-threaded program $P = \langle T, r_m \rangle$ is made up of a set of n threads $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$ where $\tau_i = \langle R_i, <_i, \mathfrak{E}_i \rangle$, R_i is a set of regions ordered under $<_i$ and r_m is the immortal region ($\mathrm{EDP}^{\mathcal{M}^{\mathrm{RPM}^{\|}}}(r_m) = \infty$) executed at the pre-mission phase. A region is denoted as r_j^i if it belongs to the set R_i of thread τ_i and is the j^{th} region in the sequence under $<_i$. As with RPM$^{\rightarrow}$, each region r_i in RPM$^{\|}$ has an EDP specified in the set $\mathfrak{E}_i$ that identifies a deallocation point that lies along the same thread of execution as that region. The LAP of every region r_j^i is $r_j^{i^-}$. In the constrained version of RPM$^{\|}$, cRPM$^{\|}$, each thread τ_i is defined as $\tau_i = \langle R_i, <_i, \mathfrak{E}_i, \mathfrak{A}_i, \mathfrak{D}_i \rangle$ as each region also has a specified RAP and RDP and for which allocation and deallocation points also lie along the same thread of execution as that region. Execution of regions proceeds as follows in both RPM$^{\|}$ and cRPM$^{\|}$: an initial thread τ_m executes the region r_m in which the immortal objects are created. A thread is then created for each of the elements τ_i in T with each thread executing each region in R_i in sequence order. Clearly, this model is similar to having a number of programs, one for each thread, as defined for RPM$^{\rightarrow}$ and cRPM$^{\rightarrow}$. Similar equations for deriving the memory requirements of these programs are derived.

Let $\Pi(P)$ be the set of programs defined in RPM$^{\rightarrow}$ so that if $P = \langle T, r_m \rangle$ in RPM$^{\|}$ then $\Pi(P) = \{P_i \langle R_i, <_i, \mathfrak{E}_i \rangle;\ \tau_i = \langle R_i, <_i, \mathfrak{E}_i \rangle \varepsilon T\}$. Similarly, $\Pi(P)$ may be the set of programs defined in cRPM$^{\rightarrow}$ so that if $P = \langle T, r_m \rangle$ in RPM$^{\|}$ then $\Pi(P) = \{P_i = \langle R_i, <_i, \mathfrak{E}_i, \mathfrak{A}_i, \mathfrak{D}_i \rangle;\ \tau_i = \langle R_i, <_i , \mathfrak{E}_i, \mathfrak{A}_i, \mathfrak{D}_i \rangle \varepsilon T\}$. Using Equations 4.3 and 4.8 respectively, the memory requirements for $\mathcal{M}^{\mathrm{RPM}^{\|}}$ and $\mathcal{M}^{\mathrm{cRPM}^{\|}}$ can be found:

$$\mathrm{MemReq}^{\mathcal{M}^{\mathrm{RPM}^{\|}}}(P) = \mid r_m \mid + \sum_{\forall\, P_i \varepsilon \Pi(P)} \mathrm{MemReq}^{\mathcal{M}^{\mathrm{RPM}^{\rightarrow}}}(P_i) \tag{4.9}$$

$$\mathrm{MemReq}^{\mathcal{M}^{\mathrm{cRPM}^{\|}}}(P) = \mid r_m \mid + \sum_{\forall\, P_i \varepsilon \Pi(P)} \mathrm{MemReq}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(P_i) \tag{4.10}$$

The assumption at this point is that the worst case memory requirements of each thread occur at the same instance in time. This is a pessimistic approach that can be addressed by extending the LSL to allow further information to be expressed in the STS that describes execution relationships between threads. This extension is described in Section 4.2.4.

Correctness of RPM$^{\|}$ and cRPM$^{\|}$ Specifications

The correctness of programs in RPM$^{\|}$ and cRPM$^{\|}$ can be easily proven using the criteria for correctness of RPM$^{\rightarrow}$ and cRPM$^{\rightarrow}$. A program P in RPM$^{\|}$ is correct if $\forall\, P_i \varepsilon \Pi(P)$, P_i is correct in RPM$^{\rightarrow}$. Similarly, a program P in cRPM$^{\|}$ is correct if $\forall\, P_i \varepsilon \Pi(P)$, P_i is correct in cRPM$^{\rightarrow}$.

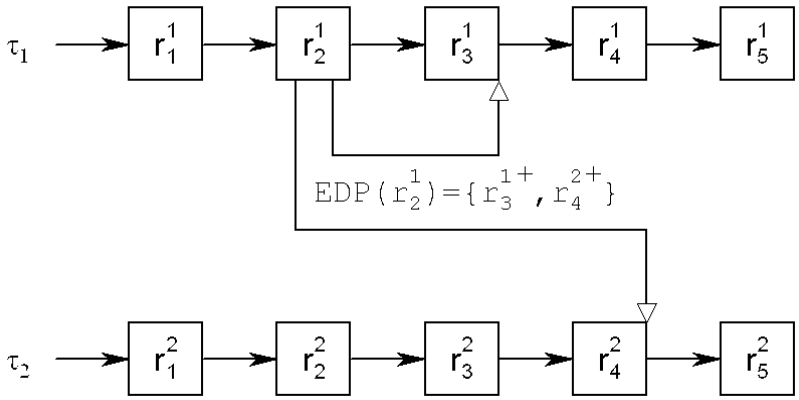


Figure 4.1: Specifying EDPs in RPM$^{\|\sigma}$

4.2.2 Multi-Threading with Sharing of Dynamic Objects: RPM$^{\|\sigma}$

In the four variations of the RPM presented so far, the EDPs, RDPs and RAPs of regions always lie along the same thread. If dangling pointers are to be avoided, then this implies that the reference graph for objects in RPM$^{\|}$ and cRPM$^{\|}$ are such that an object in region r_j^i can only reference objects in regions R_i and r_m. In order to remove this restriction, the models must be extended to define deallocation points for regions across threads, a process called *region sharing*. For example, if a thread τ_1 creates an object o_1 in region r_i^1 and sets a reference from o_1 to an object in region r_j^2 executed by thread τ_2, then enforcing referential integrity requires that r_j^2 be deallocated only after r_i^1 is deallocated. In the RPM which allows sharing of dynamic objects, RPM$^{\|\sigma}$, the LAP of a region r_j^i remains r_j^{i-} but each region must have a deallocation point specified in the EDP of that region for every thread in the program. A formal representation of RPM$^{\|\sigma}$ together with the functions for calculating memory requirements of programs in RPM$^{\|\sigma}$ follows.

In RPM$^{\|\sigma}$, a multi-threaded program is specified in the same way as for RPM$^{\|}$ but with each region having a set of deallocation points, one for each thread in the program. Therefore, for a region r_j^i, the EDP of r_j^i is specified as $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\|\sigma}}}(r_j^i) = E$ where E is a set of inter-region locations, one for each thread. If $r_l^{k+} \varepsilon E$ then r_l is the region in the thread τ_k after which the objects in r_j^i are no longer required by τ_k. For example, if $P = \langle T, r_m, \mathfrak{E} \rangle$ and $T = \{\langle \{r_1^1, r_2^1, r_3^1\}, <_1 \rangle, \langle \{r_1^2, r_2^2\}, <_2 \rangle\}$, then if r_2^2 has an object to which some object in r_3^1 holds a reference, then the EDP of r_2^2 might be specified as $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\|\sigma}}}(r_2^2) = \{r_3^{1+}, r_2^{2+}\}$. Note that there is no transitive relationship between the deallocation points specified in the EDP. For example, if r_d^{c+} is an element of $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\|\sigma}}}(r_b^a)$ and r_f^{e+} is an element of $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\|\sigma}}}(r_d^c)$, then this does not imply that r_f^{e+} is also an element of $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\|\sigma}}}(r_b^a)$. If a region does require any objects in regions of some thread τ_i, then r_1^{i-} is the only element in E. Execution in RPM$^{\|\sigma}$ is defined in the same way as that of RPM$^{\|}$.

Calculating an expression for the memory requirements in RPM$^{\|\sigma}$ requires observation of the dependencies across threads. Consider Figure 4.1[9] which shows a program ϕ with two threads and a region r_2^1 with $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\|\sigma}}}(r_2^1) = \{r_3^{1+}, r_4^{2+}\}$. As it is unknown by τ_1 when r_4^2 will be executed, the memory for r_2^1 can not be deallocated at r_3^{1+} and thus the memory requirements of τ_1 at r_4^1 and r_5^1 are increased by $\mid r_2^1 \mid$. This is tantamount to assuming that all regions with a cross-thread deallocation point specified are never deallocated. In RPM$^{\|}$, the worst case memory requirements were derived by summing the region in each thread which incurs the largest memory requirements in that thread. Therefore, the same approach would be unnecessarily pessimistic in RPM$^{\|\sigma}$ if it is assumed that these regions are never deallocated. For example, assume that $\mid r_2^1 \mid$ and $\mid r_5^1 \mid$ in comparison to all other regions in τ_1, and $\mid r_5^2 \mid$ in comparison to all other regions in τ_2 are very large for the program in Figure 4.1. Since $\mid r_5^1 \mid$ incurs the cost of $\mid r_2^1 \mid$ due to the cross-thread dependency, the memory requirements for the whole program would be considered to be the sum of the memory requirements at $\mid r_2^1 \mid$, $\mid r_5^1 \mid$ and $\mid r_5^2 \mid$. However, it is always the case when executing $\mid r_5^2 \mid$ that the memory for $\mid r_2^1 \mid$ could be deallocated, thus making for a pessimistic result. This affect of region sharing on the memory requirements of a program can be described by the Entropy Hypothesis. Knowledge of the exact lifetime of a region in a thread is lost due to region sharing, thereby resulting in additional memory requirements in comparison to a system in which region sharing is not present. However, by capturing the lifetime of a region against other control flows, it is not necessary to assume that the memory for regions in totally indeterministic. Therefore, using this information can reduce memory requirements in comparison to this pessimistic approach.

Finding the worst case memory requirements for a program in RPM$^{\|\sigma}$ can be achieved by finding the memory requirements for every possible combination of region execution across the threads. For every combination, only regions that would not be deallocated if that region combination occurs need be calculated. This approach makes for an algorithm that will fail to scale well. However, it is not necessary to calculate the memory requirements of every possible combination of region execution. Instead, the worst case requirements of partitions of each thread that may change due to cross-thread dependencies can be derived first, thereby reducing the search space. These partitions are called *memory traces* with each memory trace giving the memory requirements of executing that trace.[10] For example, assuming no other sharing between threads, the program in Figure 4.1 consists of 2 possible memory traces: τ_1 reaching r_4^1 before τ_2 reaches r_5^2 and τ_1 not reaching r_4^1 before τ_2 reaches r_5^2. Therefore, the memory requirements of this program ϕ is equal to:

$$\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\|\sigma}}}(\phi) = \quad \max(\quad \text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r^2_{\overrightarrow{1,4}}) + \tag{4.11}$$

[9]Note the two types of arrows used in this figure to describe program flow and deallocation points will be used throughout this thesis (in the latter case also to describe allocation points).

[10]Note that programs in RPM$^{\|}$ are a special case of programs in RPM$^{\|\sigma}$ with a single memory trace.

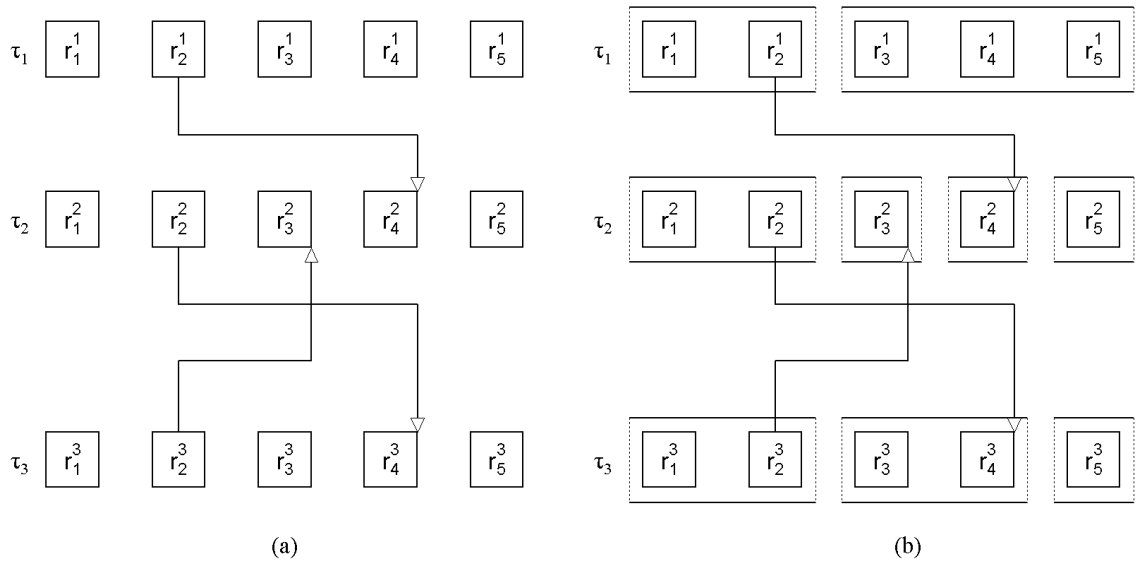


Figure 4.2: Partitioning Threads in RPM$^{\|\sigma}$

$$\begin{aligned} &\mathrm{MemReq}^{\mathcal{M}^{\mathrm{RPM}^{\rightarrow}}}({}^{*}r^{1}_{\overrightarrow{1,5}}),\\ &\mathrm{MemReq}^{\mathcal{M}^{\mathrm{RPM}^{\rightarrow}}}(r^{2}_{\overrightarrow{5,5}}) +\\ &\mathrm{MemReq}^{\mathcal{M}^{\mathrm{RPM}^{\rightarrow}}}(r^{1}_{\overrightarrow{1,5}}) \end{aligned}$$

$$)$$

where ${}^{*}r^{1}_{\overrightarrow{1,5}}$ is the program made of regions $r^{1}_{\overrightarrow{1,5}}$ but with $\mathrm{EDP}^{\mathcal{M}^{\mathrm{RPM}^{\rightarrow}}}(r^{1}_{2}) = r^{1}_{5}{}^{+}$ instead of $r^{1}_{3}{}^{+}$ in τ_1.

In order to calculate the memory requirements of programs in RPM$^{\|\sigma}$ it is therefore necessary to first identify the set of possible memory traces. This is achieved by splitting a sequence in two if it contains a cross-thread deallocation point so that the regions affected by this and those that are not are considered separately. As observed in the previous example, if a cross-thread deallocation point from a region $r^{i}_{j'}$ in thread τ_i to some other region $r^{k}_{l'}{}^{+}$ in thread τ_k exists, when considering two sequences $r^{i}_{\overrightarrow{j,j''}}$ and $r^{k}_{\overrightarrow{l,l''}}$ ($r^{i}_{j'}\varepsilon r^{i}_{\overrightarrow{j,j''}}$ and $r^{k}_{l'}\varepsilon r^{k}_{\overrightarrow{l,l''}}$) then the memory requirements are the maximum of either the sum of executing ${}^{*}r^{i}_{\overrightarrow{j,j''}}$ with $r^{k}_{\overrightarrow{l,l'}}$ or executing $r^{i}_{\overrightarrow{j,j''}}$ with $r^{k}_{\overrightarrow{l'+1,l''}}$. For simplicity when deriving an expression for the memory requirements, a proper partitioning of a region sequence is opted for. This increases the search space but does not make the memory requirements any worse. ${}^{*}r^{i}_{\overrightarrow{j,j''}}$ can be split into two: $r^{i}_{\overrightarrow{j,j'-1}}$ with ${}^{*}r^{i}_{\overrightarrow{j',j''}}$ and $r^{i}_{\overrightarrow{j,j'-1}}$ with $r^{i}_{\overrightarrow{j',j''}}$. Therefore, depending on the trace under investigation, either ${}^{*}r^{i}_{\overrightarrow{j',j''}}$ or $r^{i}_{\overrightarrow{j',j''}}$ is chosen. The memory requirements of executing $r^{i}_{\overrightarrow{j,j''}}$ and $r^{k}_{\overrightarrow{l,l''}}$ concurrently is therefore the worst of executing:

- $r^{i}_{\overrightarrow{j,j'-1}}$ with $r^{k}_{\overrightarrow{l,l'}}$,

- $r^i_{\overrightarrow{j,j'-1}}$ with $r^k_{\overrightarrow{l'+1,l''}}$,
- ${}^*r^i_{\overrightarrow{j',j''}}$ with $r^k_{\overrightarrow{l,l'}}$ or
- $r^i_{\overrightarrow{j',j''}}$ with $r^k_{\overrightarrow{l'+1,l''}}$.

As a further example, consider the program in Figure 4.2 (a) which shows a program with three threads[11] and with three inter-thread deallocation points caused by r^2_4 referencing an object in r^1_2, r^3_4 referencing an object in r^2_2 and r^2_3 referencing an object in r^3_2. Figure 4.2 (b) shows how the program in Figure 4.2 (a) is partitioned. The expression $\text{Part}(P, \tau_i)$ gives the partitioning from thread τ_i in P:

$$\begin{aligned}\text{Part}(P,\tau_i) \quad = \quad & \{r^i_{\overrightarrow{j,k}} \ ; \ \mathit{BoundaryRegion}(r^i_j, \tau_i, P) \\ & \wedge \mathit{BoundaryRegion}(r^i_k, \tau_i, P) \wedge \\ & \forall\, r^i_x \varepsilon \{r^i_x >_i r^i_j \wedge r^i_x <_i r^i_k\}, \neg \mathit{BoundaryRegion}(r^i_x, \tau_i, P)\}\end{aligned} \tag{4.12}$$

where

$$\begin{aligned}\mathit{BoundaryRegion}(r^i_j, \tau_i = \langle R_i = r^i_{\overrightarrow{1,n}}, <_i\rangle, P) \equiv \\ r^i_j = r^i_1 \vee r^i_j = r^i_n \vee \\ (r^{y+}_z \varepsilon \text{EDP}^{\mathcal{M}^{\text{RPM}\|\sigma}}(r^i_j) \wedge y \neq i \wedge r^{y+}_z \neq r^{y-}_1) \vee \\ (\exists\, \tau_g = \langle R_g, <_g\rangle \varepsilon P \wedge \exists\, r^g_h \varepsilon R_g \wedge r^{i+}_j \varepsilon \text{EDP}^{\mathcal{M}^{\text{RPM}\|\sigma}}(r^g_h) \wedge g \neq i)\end{aligned} \tag{4.13}$$

A partition by Equation 4.12 is therefore a sequence of regions for which the first and last regions in the sequence are the only boundary regions of the sequence. A boundary region is either the first or last region of a thread (Line 1 of Equation 4.13), is a region with a deallocation point in another thread (Line 2 of Equation 4.13), or is an element of the EDP set of some region in another thread (Line 3 of Equation 4.13). Therefore, in Figure 4.2 (b), $\text{Part}(P, \tau_1) = \{r^1_{\overrightarrow{1,2}}, r^1_{\overrightarrow{3,5}}\}$, $\text{Part}(P, \tau_2) = \{r^2_{\overrightarrow{1,2}}, r^2_{\overrightarrow{3,3}}, r^2_{\overrightarrow{4,4}}, r^2_{\overrightarrow{5,5}}\}$ and $\text{Part}(P, \tau_3) = \{r^3_{\overrightarrow{1,2}}, r^3_{\overrightarrow{3,4}}, r^3_{\overrightarrow{5,5}}\}$. As each thread without any inter-thread deallocation points creates a partition and each cross-thread deallocation point divides a partition into two, the number of partitions in a program with m threads and n cross-thread deallocation points is $m + (2 \times n)$. The memory requirements for the program are found by finding the maximum of the memory requirements across all possible memory traces. A trace is specified as a set of partitions, one element for each thread. For example, the trace when τ_1 is executing in the second partition, τ_2 is executing in the third partition and τ_3 is executing in the second

[11] The control flow arrows are removed to make the diagram more clear.

partition in Figure 4.2 (b) is given as $\{r^1_{\overrightarrow{3,5}}, r^2_{\overrightarrow{4,4}}, r^3_{\overrightarrow{3,4}}\}$ and the memory requirements are equal to the sum of $\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(^*r^1_{\overrightarrow{3,5}})$, $\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(^*r^2_{\overrightarrow{4,4}})$ and $\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r^3_{\overrightarrow{3,4}})$. In each case, the $*$ symbol signifies that there exists a region which must be added to the memory requirements, irrespective of where the deallocation point of this region lies in its thread. In τ_1 this refers to r^1_2 whereas in τ_2 this refers to r^2_2. Once the memory requirements of all possible traces are found, the memory requirements of the program is found by selecting the greatest value.

Let $\text{LivesOn}(r^i_{i''}, \theta)$ be all the regions before $r^i_{i'}$ which have deallocation points after $r^j_{j''}$ where $r^j_{\overrightarrow{j',j''}} \varepsilon \theta$, $r^i_{i''} \varepsilon r^i_{\overrightarrow{i',i'''}}$ and $\theta \bigcup \{r^i_{\overrightarrow{i',i'''}}\}$ is a trace. For example in Figure 4.2, $\text{LivesOn}(r^1_4, \{r^2_{\overrightarrow{4,4}}, r^3_{\overrightarrow{3,4}}\}) = \{r^1_2\}$ meaning that r^1_2 can not be deallocated when τ_1 is executing r^1_4, τ_2 is executing r^2_4 and τ_3 is executing regions in $r^3_{\overrightarrow{3,4}}$, irrespective of where the deallocation point of r^1_4 falls in τ_1. The memory requirements for a region r^i_j when executing under trace θ is similar to that in Equation 4.1 but includes the size of regions in $\text{LivesOn}(r^i_j, \theta)$:

$$\begin{aligned} \text{MemReq}^{\mathcal{M}^{\text{RPM}\|\sigma}}(r^i_j, \theta) \quad = \quad & \sum\{|\ r^i_x\ |\ ;\ (r^i_x \leq_i r^i_j \wedge \\ & {r^i_a}^+ \varepsilon \text{EDP}^{\mathcal{M}^{\text{RPM}\|\sigma}}(r^i_x) \wedge {r^i_a}^+ \geq_i {r^i_j}^+) \vee \\ & r^i_x \varepsilon \text{LivesOn}(r^i_j, \theta)\} \end{aligned} \tag{4.14}$$

The memory requirements for the whole partition under trace θ is therefore the maximum of the requirements of each region in the partition:

$$\begin{aligned} \text{MemReq}^{\mathcal{M}^{\text{RPM}\|\sigma}}(r^i_{\overrightarrow{j,k}}, \theta) \quad = \max(\text{MemReq}^{\mathcal{M}^{\text{RPM}\|\sigma}}(r^a_b, \theta))\ ; \\ \forall\, r^a_b \varepsilon r^i_{\overrightarrow{j,k}} \end{aligned} \tag{4.15}$$

The memory requirements of the whole trace θ is the sum of the maximum memory requirements of each partition:

$$\begin{aligned} \text{MemReq}^{\mathcal{M}^{\text{RPM}\|\sigma}}(\theta) \quad = \sum\{ \quad & \text{MemReq}^{\mathcal{M}^{\text{RPM}\|\sigma}}(r^i_{\overrightarrow{i',i''}}, \theta')\ ; \\ & \theta' = \theta - \{r^i_{\overrightarrow{i',i''}}\} \wedge \{r^i_{\overrightarrow{i',i''}}\} \varepsilon \theta \\ \} & \end{aligned} \tag{4.16}$$

The set of all possible traces to be checked is given by the cross-product of each $\text{Part}(P, \tau_i)$ for all threads τ_i in the program. Therefore the memory requirements of a program in $\mathcal{M}^{\text{RPM}\|\sigma}$ are given by first finding the set of partitions for each thread, computing the cross product to derive the set of possible traces and then choosing the maximum of the memory requirements of these traces:

$$\begin{aligned}\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\|\sigma}}}(P = \langle T = \{\tau_1, \tau_2, \ldots, \tau_n\}, r_m\rangle) = \\ \max(\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\|\sigma}}}(\theta)) + r_m; \\ \forall\, \theta \varepsilon \text{Part}(P, \tau_1) \times \text{Part}(P, \tau_2) \times \ldots \times \text{Part}(P, \tau_n)\end{aligned} \tag{4.17}$$

4.2.3 The Constrained RPM$^{\|\sigma}$: cRPM$^{\|\sigma}$

cRPM$^{\|\sigma}$ is similar to RPM$^{\|\sigma}$ with the difference that threads have a set of RAPs and RDPs specified for the regions in that thread ($\tau_i = \langle R_i, <_i, \mathfrak{E}_i, \mathfrak{A}_i, \mathfrak{D}_i\rangle$). $\mathfrak{D}_i$ has the same format as that of $\mathfrak{E}_i$ in RPM$^{\|\sigma}$ in order that a region can specify a deallocation point on every thread. $\mathfrak{A}_i$ however has the same format as that of $\mathfrak{A}$ in cRPM$^{\rightarrow}$ as every region can only have one allocation point and this must reside in the same thread as that region's. This is the only way to deterministically guarantee that memory for a region has been allocated before it begins execution. Partitioning in cRPM$^{\|\sigma}$ can be carried out in the same way as in RPM$^{\|\sigma}$ but with partitioning being based on regions' RDPs rather than their EDPs. The memory requirements are also calculated in the same way with Equation 4.14 becoming:

$$\begin{aligned}\text{MemReq}^{\mathcal{M}^{\text{cRPM}^{\|\sigma}}}(r_j^i, \theta) \;=\; & \sum\{|\, r_x^i \mid \;;\; (r_x^i \leq_i r_j^i \wedge \\ & {r_a^i}^+ \varepsilon \text{RDP}^{\mathcal{M}^{\text{cRPM}^{\|\sigma}}}(r_x^i) \wedge {r_a^i}^+ \geq_i {r_j^i}^+) \vee \\ & (r_x^i >_i r_j^i \wedge \text{RAP}^{\mathcal{M}^{\text{cRPM}^{\|\sigma}}}(r_x^i) \leq_i {r_j^i}^-) \vee \\ & (r_x^i \varepsilon \text{LivesOn}(r_j^i, \theta))\}\end{aligned} \tag{4.18}$$

As was done with Equation 4.5, this can be reduced to:

$$\begin{aligned}\text{MemReq}^{\mathcal{M}^{\text{cRPM}^{\|\sigma}}}(r_j^i, \theta) \;=\; & \sum\{|\, r_x^i \mid \;; \\ & (({r_a^i}^+ \varepsilon \text{RDP}^{\mathcal{M}^{\text{cRPM}^{\|\sigma}}}(r_x^i) \wedge {r_a^i}^+ \geq_i {r_j^i}^+) \wedge \\ & \text{RAP}^{\mathcal{M}^{\text{cRPM}^{\|\sigma}}}(r_x^i) \leq_i {r_j^i}^-) \vee \\ & (r_x^i \varepsilon \text{LivesOn}(r_j^i, \theta))\}\end{aligned} \tag{4.19}$$

Execution in cRPM$^{\|\sigma}$ is defined in the same way as that of RPM$^{\|\sigma}$.

Correctness of RPM$^{\|\sigma}$ and cRPM$^{\|\sigma}$ Specifications

The correctness of programs in RPM$^{\|\sigma}$ and cRPM$^{\|\sigma}$ is similar to that of RPM$^{\rightarrow}$ and cRPM$^{\rightarrow}$ with the difference that every EDP and RDP of a region must have a deallocation point specified in all other threads in the program. This is required to fulfill the correctness criteria described on page 79

but where a deallocation "point" is interpreted as a condition for a region to be deallocated rather than an inter-region location. Therefore, a program $P = \langle T = \{\tau_1, \tau_2, \ldots, \tau_n\}, r_m\rangle$ in RPM$^{\|\sigma}$ is correct if for all regions r_j^i, EDP$^{\mathcal{M}^{\text{RPM}^{\|\sigma}}}(r_j^i)$ is specified once, EDP$^{\mathcal{M}^{\text{RPM}^{\|\sigma}}}(r_j^i) = \{r_{1'}^{1}{}^{+}, r_{2'}^{2}{}^{+}, \ldots, r_{n'}^{n}{}^{+}\}$, every region in this set is a valid region in the program and $r_x^{i}{}^{+} >_i r_j^i$. Each element r_h^{g+} in the EDP set except for $r_x^{i}{}^{+}$ can be replaced with τ_1^{g-} if no cross thread dependency exists between threads i and g at r_j^i. In cRPM$^{\|\sigma}$, correctness is similarly defined but with EDP$^{\mathcal{M}^{\text{RPM}^{\|\sigma}}}$ being replaced by RDP$^{\mathcal{M}^{\text{cRPM}^{\|\sigma}}}$ and the RAP of each region defined as in cRPM$^{\rightarrow}$. Also, the invariant RAP$^{\mathcal{M}^{\text{cRPM}^{\|\sigma}}}(r_j^i) \leq_i \text{LAP}(r_i) = r_i^-$ must hold in order to fulfill Step (2c) of the correctness criteria. Similarly, for all $r_{x'}^{x}{}^{+}\varepsilon$EDP$^{\mathcal{M}^{\text{RPM}^{\|\sigma}}}(r_j^i)$ and $r_{y'}^{x}{}^{+}\varepsilon$RDP$^{\mathcal{M}^{\text{cRPM}^{\|\sigma}}}(r_j^i)$, $r_{x'}^{x}{}^{+} \leq_i r_{y'}^{x}{}^{+}$ fulfills Step (3c).

4.2.4 Multi-Threading with Region Synchronisation: RPM$^{\|\sigma\rho}$ and cRPM$^{\|\sigma\rho}$

Multi-threaded programs often require synchronisation between threads. This can be achieved by a number of techniques such as monitors, locks and rendezvous. In the RPM variations considered so far, synchronisation is transparent to the memory model and could still be implemented in programs. However, by exposing synchronisation to the memory model it is possible to further reduce the worst case memory requirements of a program. This is a manifestation of the Entropy Hypothesis described in Chapter 3.

In RPM$^{\|\sigma}$, references across threads have an impact on the worst case memory requirements of programs but do not in any way change the indeterminacy of executing programs. This means that the model assumes that if an object needs to reference another object in another thread which has not yet been created, then it is assumed that the reference is not required and execution can continue in both threads. This is often not the case and threads may sometimes need to pause their execution in order for other threads to create the objects they require.

In RPM$^{\|\sigma\rho}$, a program $P = \langle T, r_m, H\rangle$ is defined in the same way as for RPM$^{\|\sigma}$ but with the added parameter H that defines the set of synchronisation points in the program. H is a set of pairs $(r_{i'}^i, r_{j'}^j)$ $(i \neq j)$ that signifies that region $r_{i'}^i$ can not begin execution until $r_{j'}^j$ has completed executing. Apart from this difference, execution in RPM$^{\|\sigma\rho}$ is defined in the same way as that of RPM$^{\|\sigma}$.

The memory requirements of programs in RPM$^{\|\sigma\rho}$ must take into consideration the possible reachable *execution traces* in the program's execution. Consider Figure 4.3 which shows a program ϕ with two threads and with a synchronisation point (r_2^1, r_4^2). No region in $r_{\overrightarrow{2,5}}^1$ can ever be executing while τ_2 is executing regions in $r_{\overrightarrow{1,4}}^2$. Therefore, the memory requirements of ϕ assuming no cross thread dependencies of the type of RPM$^{\|\sigma}$ are:

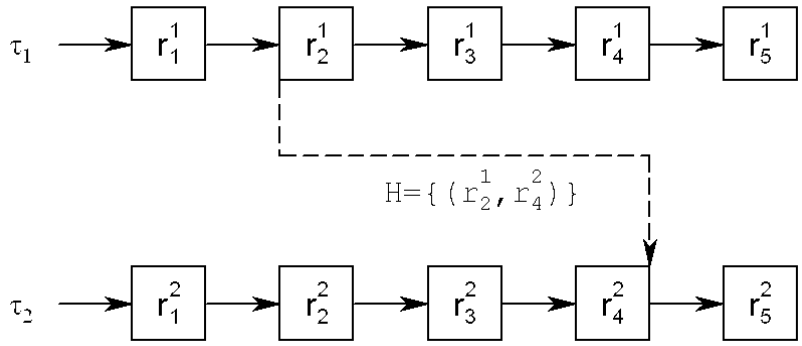


Figure 4.3: An Example of a Program in RPM$^{\|\sigma\rho}$

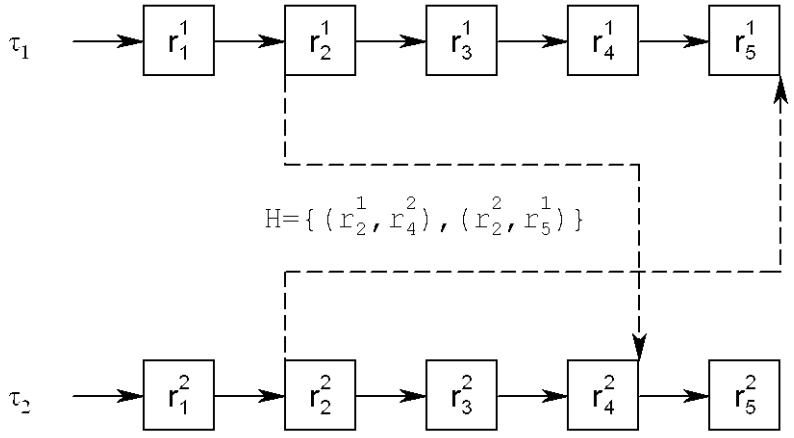


Figure 4.4: Cyclic Dependencies in RPM$^{\|\sigma\rho}$

$$\begin{aligned}\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\|\sigma\rho}}}(\phi) = \quad \max(\quad & \text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r^1_{\overrightarrow{1,1}}) + \\ & \text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r^2_{\overrightarrow{1,5}}), \\ & \text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r^1_{\overrightarrow{2,5}}) + \\ & \text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r^2_{\overrightarrow{5,5}})\end{aligned} \tag{4.20}$$

In order to derive the memory requirements of programs in RPM$^{\|\sigma\rho}$ it is first necessary to find the set of possible execution traces. An execution trace is denoted in the same way as a memory trace. In the program in Figure 4.3, there are two execution traces: $\{r^1_{\overrightarrow{1,1}}, r^2_{\overrightarrow{1,5}}\}$ and $\{r^1_{\overrightarrow{2,5}}, r^2_{\overrightarrow{5,5}}\}$. Before deriving the set of possible execution traces in a program, two observations are made on the H set of a program. Consider Figure 4.4 in which $H = \{(r^1_2, r^2_4), (r^2_2, r^1_5)\}$. Clearly, this program will experience deadlock due to a cycle of dependencies: r^1_2 can not begin executing until r^2_4 is executed which, by definition of execution in RPM$^{\|\sigma\rho}$, can not begin executing until r^2_2 is executed; r^2_2 can

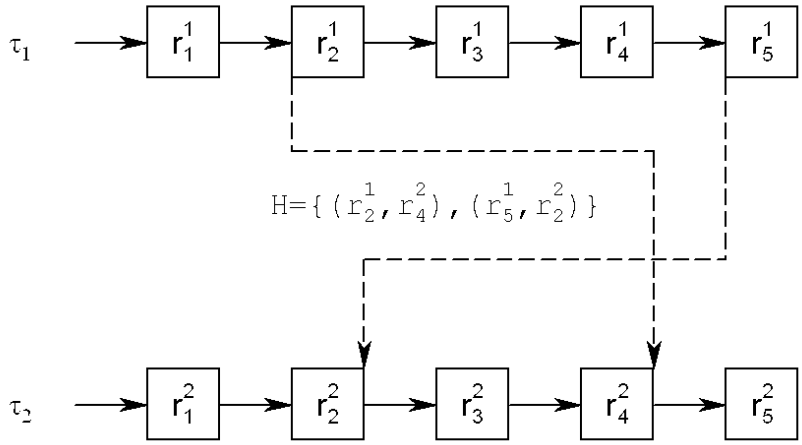


Figure 4.5: Redundant Dependencies in RPM$^{\|\sigma\rho}$

not execute until r_5^1 is executed which in turn can not be executed until r_2^1 is executed. Finding the set of possible execution traces (as well as correctness) will assume the non-existence of deadlock. Finding deadlock can be achieved using traditional techniques of finding cycles of dependencies. Let the dependency graph of program $P = \langle T, r_m, H\rangle$ be the directed graph $G(P) = \langle R, E\rangle$ where R is a set of regions that includes all the regions in all threads in the program P and E is a set of edges such that $(r_j^i, r_l^k)\varepsilon E$ if $(r_j^i, r_l^k)\varepsilon H$ or $i = k \wedge j = \text{pred}(l)$ under $<_i$. If $G(P)$ is cyclic then deadlock will occur when executing P and therefore the STS of P is determined to be incorrect. The second observation relates to redundant dependencies in the H set. Consider Figure 4.5 in which $H = \{(r_2^1, r_4^2), (r_5^1, r_2^2)\}$. As the execution of r_5^1 depends on that of r_2^1 based on the definition of execution in RPM$^{\|\sigma\rho}$, r_2^1 depends on r_4^2 from H and r_4^2 depends on that of r_2^2, the element (r_5^1, r_2^2) in H is redundant. The set H can therefore be minimised so that no similar redundant pairs exists by taking advantage of the transitivity of the dependency relation. Given the acyclic dependency graph of program $G(P)$ and the set H of P, if (r_j^i, r_l^k) is in H and there is an alternative path from r_j^i to r_l^k in $G(P)$ then (r_j^i, r_l^k) can be removed from H. Note that this implies that a region can only synchronise with one other region on another thread.

Assuming a program $P = \langle T, r_m, H\rangle$ has no cycles in $G(P)$ and H is minimised, the set of possible execution traces of P can be found by eliminating for each element of H the combinations of region executions that are impossible to reach. Rather than deriving an expression in set notation as done so far, an algorithm to achieve this is devised immediately. The algorithm is initialised with the single trace $\{r^1_{\overrightarrow{1,\sharp 1}}, r^2_{\overrightarrow{1,\sharp 2}}, \ldots, r^n_{\overrightarrow{1,\sharp n}}\}$ where n is the number of threads in P and $r^i_{\sharp i}$ is the last element of thread i. Each element from the set H creates a new set of possible traces with the algorithm terminating when H is empty. The complete algorithm is shown in Algorithm 1. As an example of the operation of this Algorithm, consider the program in Figure 4.6. In this program,

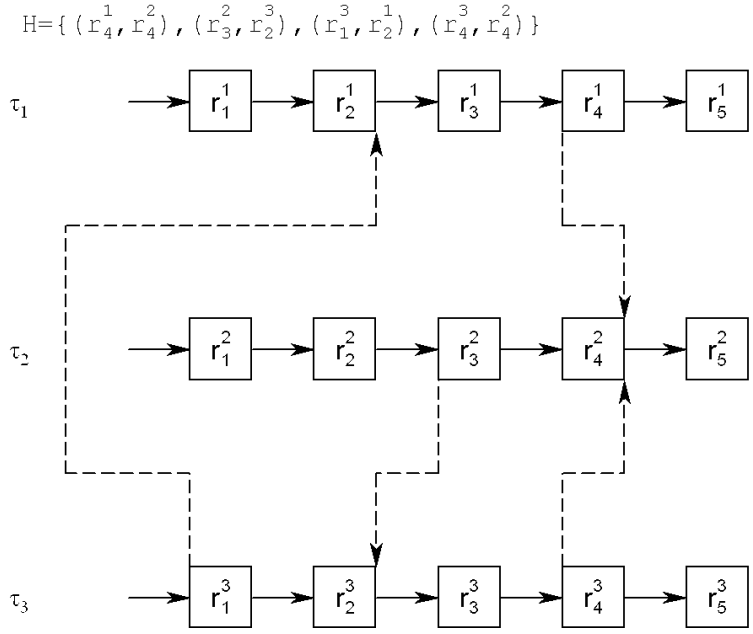


Figure 4.6: Multiple Dependencies in RPM$^{\|\sigma\rho}$

$H = (r_4^1, r_4^2), (r_3^2, r_2^3), (r_1^3, r_2^1), (r_4^3, r_4^2)$ and the execution of the algorithm proceeds as shown in Table 4.1 with every element removed from the set H expanding the rows marked with $*$. The number of possible concurrent executions is therefore reduced from 125 to 55 in this example.

Having derived the set of execution traces in a program, it is now possible to find the worst case memory requirements of programs in RPM$^{\|\sigma\rho}$ by using Equation 4.16 and inserting every execution trace for the parameter θ in this equation. However, as was shown in RPM$^{\|\sigma}$, the worst case memory requirements of this approach are pessimistic if the EDPs of regions are not considered. Therefore, the execution traces of a program can be further partitioned to create the final set of traces. This is done by first further partitioning execution traces into memory traces using Equation 4.12 where for an execution trace $\theta = \{r^1_{\overrightarrow{1',1''}}, r^2_{\overrightarrow{2',2''}}, \ldots, r^n_{\overrightarrow{n',n''}}\}$, $P = \langle T = \{\tau_1 = r^1_{\overrightarrow{1',1''}}, \tau_2 r^1_{\overrightarrow{2',2''}}, \ldots, \tau_n r^n_{\overrightarrow{n',n''}}\}, r_m\rangle$. The memory requirements for a region can then be found using Equation 4.14 and Equation 4.19 for programs in RPM$^{\|\sigma\rho}$ and cRPM$^{\|\sigma\rho}$ respectively. The memory requirements of a trace are then found using Equation 4.16 and the largest of the cross-product in Equation 4.17 gives the memory requirements of programs in RPM$^{\|\sigma\rho}$ and cRPM$^{\|\sigma\rho}$ accordingly.

Correctness of RPM$^{\|\sigma\rho}$ and cRPM$^{\|\sigma\rho}$ Specifications

The correctness of programs in RPM$^{\|\sigma\rho}$ is the same as that in RPM$^{\|\sigma}$ with the added requirement that all cross-thread deallocation points specified in an EDP fall within a possible execution trace. This is only required as a sanity check since the cross-thread deallocation point would otherwise

	X	I
$H = \{(r_4^1, r_4^2), (r_3^2, r_2^3), (r_1^3, r_2^1), (r_4^3, r_4^2)\}$	$\{\{r^1_{\overrightarrow{1,5}}, r^2_{\overrightarrow{1,5}}, r^3_{\overrightarrow{1,5}}\}\}$	*
$H = \{(r_3^2, r_2^3), (r_1^3, r_2^1), (r_4^3, r_4^2)\}$	$\{\{r^1_{\overrightarrow{1,3}}, r^2_{\overrightarrow{1,5}}, r^3_{\overrightarrow{1,5}}\}, \{r^1_{\overrightarrow{4,5}}, r^2_{\overrightarrow{5,5}}, r^3_{\overrightarrow{1,5}}\}\}$	*
$H = \{(r_1^3, r_2^1), (r_4^3, r_4^2)\}$	$\{\{r^1_{\overrightarrow{1,3}}, r^2_{\overrightarrow{1,2}}, r^3_{\overrightarrow{1,5}}\}, \{r^1_{\overrightarrow{1,3}}, r^2_{\overrightarrow{3,5}}, r^3_{\overrightarrow{3,5}}\}, \{r^1_{\overrightarrow{4,5}}, r^2_{\overrightarrow{5,5}}, r^3_{\overrightarrow{1,5}}\}\}$	*
$H = \{(r_4^3, r_4^2)\}$	$\{\{r^1_{\overrightarrow{3,5}}, r^2_{\overrightarrow{1,2}}, r^3_{\overrightarrow{1,5}}\}, \{r^1_{\overrightarrow{1,3}}, r^2_{\overrightarrow{3,5}}, r^3_{\overrightarrow{3,5}}\}, \{r^1_{\overrightarrow{4,5}}, r^2_{\overrightarrow{5,5}}, r^3_{\overrightarrow{1,5}}\}\}$	* *
$H = \varnothing$	$\{\{r^1_{\overrightarrow{3,5}}, r^2_{\overrightarrow{1,2}}, r^3_{\overrightarrow{1,5}}\}, \{r^1_{\overrightarrow{1,3}}, r^2_{\overrightarrow{3,5}}, r^3_{\overrightarrow{3,3}}\}, \{r^1_{\overrightarrow{1,3}}, r^2_{\overrightarrow{5,5}}, r^3_{\overrightarrow{4,5}}\}, \{r^1_{\overrightarrow{4,5}}, r^2_{\overrightarrow{5,5}}, r^3_{\overrightarrow{1,5}}\}\}$	

Table 4.1: Executing *GetExecutionTraces()* for the program in Figure 4.6

Data : $P = \langle T = \{\tau_1, \ldots, \tau_n\}, r_m, H\rangle$
Result: X: Set of Execution Traces
X = $\{\{r^1_{\overrightarrow{1,\sharp 1}}, r^2_{\overrightarrow{1,\sharp 2}}, \ldots, r^n_{\overrightarrow{1,\sharp n}}\}\}$;
while $H! = \{\}$ **do**
  $H = H - \{(r_j^i, r_l^k)\}$; where $(r_j^i, r_l^k) \varepsilon H$
  forall $I = \{r^1_{\overrightarrow{1',1''}}, \ldots, r^i_{\overrightarrow{i',i''}}, \ldots, r^k_{\overrightarrow{k',k''}} \ldots, r^n_{\overrightarrow{n',n''}}\} \varepsilon X$ **do**
    if $r^i_{i'} \leq_i r^i_j \leq_i r^i_{i''} \wedge r^k_{k'} \leq_k r^k_l \leq_k r^k_{k''}$ **then**
      A1 = A2 = $\varnothing$;
      if $r^i_j >_i r^i_{i'}$ **then**
        A1 = $\{r^1_{\overrightarrow{1',1''}}, \ldots, r^i_{\overrightarrow{i',j-1}}, \ldots, r^k_{\overrightarrow{k',k''}} \ldots, r^n_{\overrightarrow{n',n''}}\}$;
      end
      if $r^k_l <_k r^k_{k''}$ **then**
        A2 = $\{r^1_{\overrightarrow{1',1''}}, \ldots, r^i_{\overrightarrow{j,i''}}, \ldots, r^k_{\overrightarrow{l,k''}} \ldots, r^n_{\overrightarrow{n',n''}}\}$
      end
      X = X−I $\bigcup$ {A1,A2};
    end
  end
end

Algorithm 1: GetExecutionTraces($P = \langle T = \{\tau_1, \ldots, \tau_n\}, r_m, H\rangle$

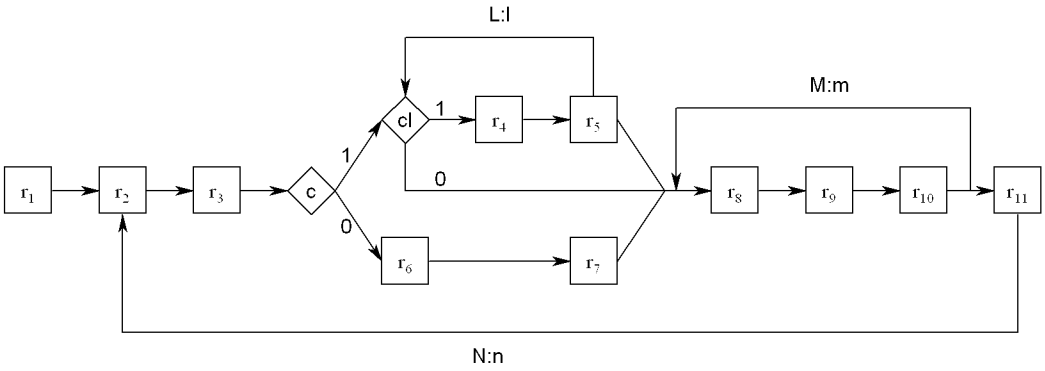


Figure 4.7: An Example of a Program with All Control Flows

be redundant. For example in Figure 4.3, if $\text{EDP}^{\mathcal{M}^{\text{RPM}\|\sigma\rho}}(r_4^1) = \{r_4^{1+}, r_2^{2+}\}$, then the deallocation point r_2^{2+} is redundant since by the time r_4^1 begins executing, r_2^2 is guaranteed to have already finished executing. The correctness of programs in $\text{cRPM}^{\|\sigma\rho}$ is also the same as that in $\text{cRPM}^{\|\sigma}$ with a similar additional requirement; that is that deallocation points specified in an RDP must fall within a possible execution trace.

4.3 Iterative Loops and Conditionals in the Region Partitioning Model

A sequential and finite description of a program is too restrictive for real-time applications, particularly as tasks in a real-time program are typically described as periodic activities enclosed in an infinite loop. It is therefore necessary to consider how iteration and selection can be described in the RPM. Consideration is given to both iteration and selection that occurs entirely within a region (intra-region control flow) or encapsulates a number of regions in their entirety (inter-region control flow).

Recall that a region is a coarse grain entity that specifies that all objects created within that region are to have an equal lifetime as specified by the EDP of that region. Therefore, intra-region control flow is not a concern within the RPM as no lifetime information is specified for memory usage at any finer granularity than the region. For each region, it is only necessary to know the memory requirements for executing that region. In order to guarantee finite memory requirements, all intra-region loops must either be bounded by a fixed number of iterations or, for infinite loops, only a finite number of iterations can create new objects.

In order to describe new inter-region control flows in the extended RPM, eRPM, the program definition is extended to that shown in Table 4.2. This program definition ensures that loops and conditionals are properly nested and goto-like branching is forbidden. The main control structures defined are sequential regions, loops with a fixed number of iterations (fixed loops), loops that

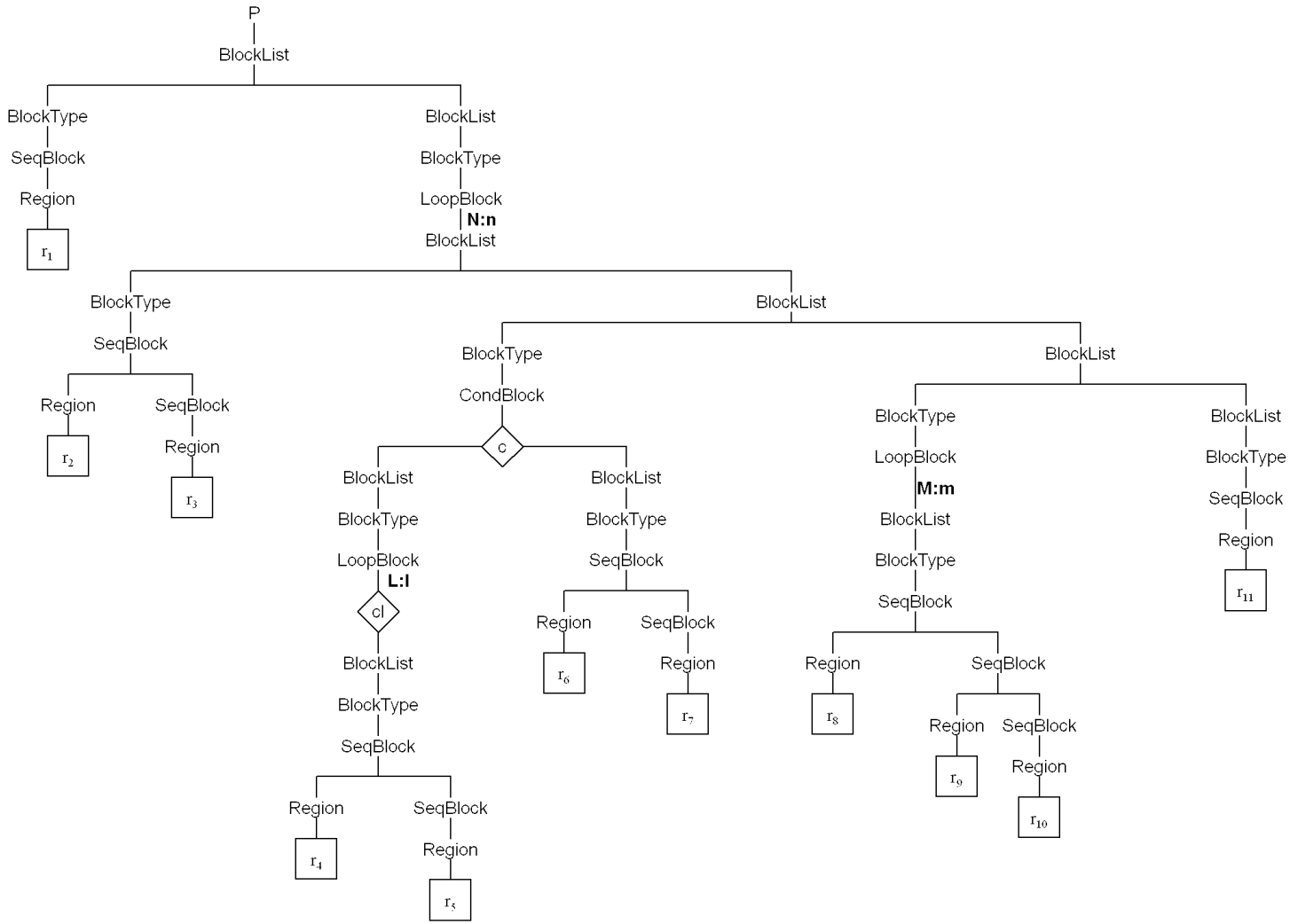


Figure 4.8: Specification of Program in Figure 4.7

```
<P>          :: =  <BlockList>
<BlockList>  :: =  <BlockType> | <BlockType>,<BlockList>
<BlockType>  :: =  <SeqBlock> | <LoopBlock> | <CondBlock>
<SeqBlock>   :: =  Region | Region,<SeqBlock>
<LoopBlock>  :: =  (int,<BlockList>) | ∞,<BlockList>) |
                   (Condition,int,<BlockList>) |
                   (Condition, ∞ ,<BlockList>)
<CondBlock>  :: =  (Condition,<BlockList>,<BlockList>)
```

Table 4.2: Program Definition in eRPM

never terminate (infinite loops), loops that have a condition that specifies whether to carry out another iteration but for which a maximum number of iterations is specified (conditional loops), conditional loops for which the maximum number of iterations is unspecified (infinite&conditional loops)[12] and if-then-else style conditionals. As conditional checks may involve code execution that creates objects, conditionals could be considered as executable blocks with memory requirements and an allocation and deallocation point in the same way as a regular region. This will be the case for conditional regions in conditional branches as well as for regions in conditional loops. It is noted that the conditional region of a conditional loop can execute for one more iteration than the maximum number of iterations. Indeed, if a conditional loop is reached, the conditional region of that loop always executes for one more iteration than the regions in the loop. In eRPM, as in the previously described versions of the RPM, each region must specify allocation and deallocation points from which the correctness criteria and memory requirements of the program can be derived.

The sequential execution of regions in RPM$^{\rightarrow}$ and cRPM$^{\rightarrow}$ made it possible to define a total ordering on regions using $<$ on which execution was defined. This is no longer possible in eRPM as conditional branching creates a partial order of program flow. Therefore, unlike the previous RPM models which required the ordering of regions to form part of the program specification, the ordering of regions in eRPM is implicit in the specification of the program using the BNF notation in Table 4.2. Hence, a program execution in eRPM is defined by a control flow graph (CFG) such as that in Figure 4.7 which illustrates a program with an if-then-else conditional statement, two fixed loops and one conditional loop. Each loop is given a name - L, M and N in this case. The letters m and n denote the number of iterations for fixed loops. The letter l denotes the known worst case number of iterations for the conditional loop. The diamond symbol denotes a conditional region; c_1 is the condition tested in the if-then-else conditional and $c1$ is that tested in the conditional loop L. Figure 4.8 gives a tree representation of the program in Figure 4.7 derived using this new program description. Assume a program is defined using regions for which indexes are text identifiers rather than integers (this is simply to avoid confusion with the total ordering of integers). Therefore, a program P might include the regions r_{start}, r_b, r_c, and so on until r_{exit}. The

[12]This terminology for differentiating between the four types of loops is used extensively throughout this thesis.

starting region is identified by the name r_{start}. A program P may have more than one exit region but a new empty region r_{exit} can be introduced which follows the execution of every terminating region from the original program. The CFG is generated from the program description using the Algorithms given in Appendix A. CFG(P) is a tuple $\langle R, E, r_{start}, r_{end} \rangle$ where R describes the regions of P, E describe the edges of the CFG and r_{start} and r_{end} ($r_{start}, r_{end} \varepsilon R$) are the start and end regions respectively. Therefore, R is a set of pairs (*RegionName*, *Type*) where *RegionName* is a string identifier for the region and $Type \varepsilon \{SimpleRegion, ConditionalRegion\}$. Edges are annotated with the type of edge, that is whether it is a forward edge or a back edge from a loop and, in the case of a back edge, the number of iterations in the loop. Edges exiting conditional regions are also annotated in this way together with "1" or "0" to signify the edge taken as a result of the conditional operation. Therefore, E is a set of tuples ($R1, R2$, *Type*, *CondResult*, *Iterations*) where $\{R1, R2\} \varepsilon R$, $Type \varepsilon \{Forward, Back, Conditional\}$, $CondResults \varepsilon \{1, 0\}$ and $Iterations \varepsilon \mathbb{Z}^+ \bigcup \{\infty\}$. Note that when *Type* is *Forward*, *CondResult* and *Iterations* are ignored. When *Type* is *Back*, *CondResult* is ignored but *Iterations* holds the number of iterations of the loop; the total number of iterations for a fixed loop and the maximum number of iterations for a conditional loop. When *Type* is *Conditional*, *Iterations* is ignored whereas the value of *CondResult* is "1" for the edge when the result of the conditional check is `true` and "0" when it is `false`.

In addition to being used to define program execution in RPM$^{\rightarrow}$, the total order $<$ was also used to define the correctness criteria of EDPs and RAPs in an STS and to derive the expressions for the memory requirements of programs. A new ordering for regions is introduced that is derived using the CFG is used to achieve these goals in eRPM. The first step is to eliminate back edges from the CFG. While it is possible to find loops in a CFG using one of several algorithms [7], the implicitly structured program definition in Table 4.2 means that it is simpler to modify Algorithms given in Appendix A so that no back edges are created by the algorithm. Next, the edges representing the result of a conditional check for a conditional loop are also removed. This acyclic graph is called CFG$^{NoLoop}(P)$. A relationship $\preceq$ is introduced so that if $r_i \preceq r_j$, then there exists a path in CFG$^{NoLoop}(P)$ between r_i and r_j or $r_i = r_j$. Likewise, $\succeq$, $\prec$ and $\succ$ are introduced. The regions in P ordered under $\preceq$ form a lattice with r_{start} as the minimum element and r_{exit} as the maximum element[13]. Let R^c be the set of regions of P unioned with $\overline{c}$ for every conditional region c or a conditional branch (not of a conditional loop) in R. Let E^c be the set of edges of P but replacing (c, r) with $(\overline{c}, r)$ for every edge (c, r) in E which is annotated with "0" to indicate the result of c in a conditional branch being `false`. The set of *Tracks* of a program P, *Tracks*(P) is the set of all the possible paths from r_{start} to r_{exit} in CFG$^{NoLoop}(P)$ with these modifications:

[13]Note that this lattice and CFG$^{NoLoop}(P)$ are isomorphic.

$$
\begin{aligned}
\text{Tracks}(P) \quad = \quad & \{\rho = \langle r, e, r_{start}, r_{end}\rangle; \\
& \text{CFG}^{NoLoop}(P) = \langle R, E, r_{start}, r_{end}\rangle \wedge \\
& r \varepsilon \mathbb{P}(R^c), e \varepsilon \mathbb{P}(E^c) \wedge r_{start} \varepsilon r \wedge \\
& \forall\, r_i \varepsilon r - \{r_e\}\, \exists\, r_i' \varepsilon r \wedge (r_i, r_i') \varepsilon E^c \\
& \wedge \forall\, r_1, r_2 \varepsilon r, r_1 \preceq r_2 \vee r_2 \preceq r_1 \\
& \wedge (r_1, r_2) \varepsilon E^c \Rightarrow (r_1, r_2) \varepsilon e\}
\end{aligned} \tag{4.21}
$$

Note that regions include both regular and conditional regions. In the latter case, if the track is formed by the conditional result or a conditional branch being `true`, then the notation c is used to identify that region. If it is formed by the conditional result of a conditional branch being `false`, then the notation $\overline{c}$ is used. It is emphasised that conditional loops are treated as fixed loops in that the edge of the conditional branch with "0" is removed. For example, the two tracks in Figure 4.7 on page 93 contain the regions $\{r_1, r_2, r_3, c, c1, r_4, r_5, r_8, r_9, r_{10}, r_{11}\}$ and $\{r_1, r_2, r_3, \overline{c}, r_6, r_7, r_8, r_9, r_{10}, r_{11}\}$ and with a single edge between each region in the order as given here. By separating conditional regions into c and $\overline{c}$ for conditional branches, the regions of each track in Tracks(P) form a total order since $\forall\, r_1, r_2 \varepsilon r, r_1 \preceq r_2 \vee r_2 \preceq r_1$. Therefore, regions in a track can be ordered using $<$ so that if $\rho \varepsilon \text{Tracks}(P)$ and R is the set of regions in ρ, then if $r_i < r_j$ where $r_i, r_j \varepsilon R$, there exists a path in ρ between r_i and r_j and $r_i \neq r_j$. Similarly, $>$, $\leq$ and $\geq$ are defined for regions in a track.

Inter-region locations are introduced for describing allocation and deallocation points in the same way as for programs in RPM$^{\rightarrow}$. Therefore, if r_i is a region in the program, then the inter-region locations r_i^- and r_i^+ can be introduced to mark the point before r_i is executed where memory can be allocated and the point after r_i where memory can be deallocated. When considering a track, the $<$ ordering can be extended to use these inter-region locations in the same way as in RPM$^{\rightarrow}$ so that if $(r_i, r_j) \varepsilon E$, then $r_i^- < r_i < r_i^+ < r_j^- < r_j < r_j^+$ on that track. Finally, the inter-region location following the result of a conditional loop returning `false` needs to be considered. For every conditional loop with condition c, c^+ and c^- are already defined using the previously approach but $\overline{c}^+$ and $\overline{c}^-$ are not. The inter-region location $\overline{c}^-$ does not exist as the result of the conditional is not known at this time. However, $\overline{c}^+$ needs to be defined in order to identify the point after c when it returns `false` and regions can be deallocated. Therefore the regions deallocated after c evaluates to `false` may be different to those deallocated with c evaluates to `true`. The order of $\overline{c}^+$ in relation to other regions does not have to be defined as it will always be treated as a special case. Note that if a conditional loop is reached, the first inter-region location will be c^- of the first iteration of the loop. $\overline{c}^+$ will also be guaranteed to be reached once the conditional loop exits or

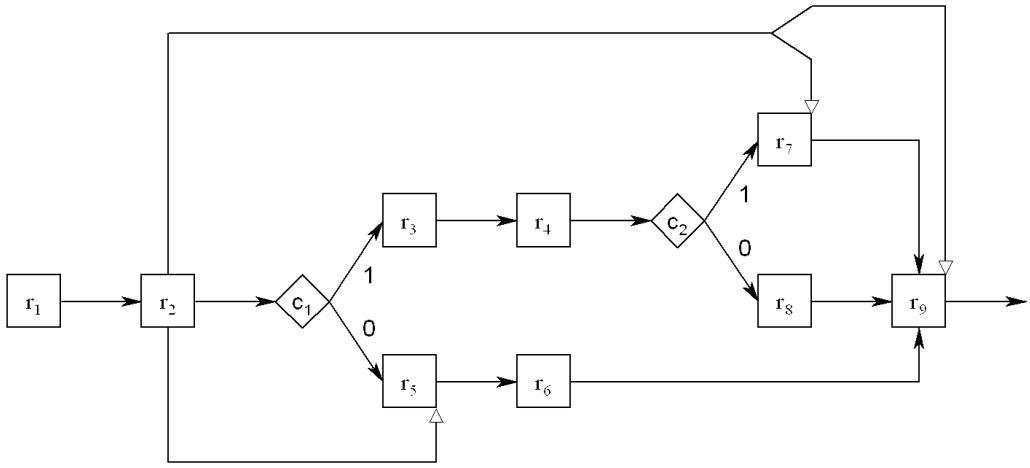


Figure 4.9: An Example of a Program with Two Conditional Branches

the maximum number of iterations is reached. Also, if the i^{th} iteration of the conditional region c of a conditional loop returns `true`, the inter-region location c^- of the $i^{th}+1$ iteration of the loop is also guaranteed to be reached. The importance of these observations will become apparent when specifying the correctness criteria for programs with conditional loops.

4.3.1 Conditional Flows in the RPM: RPM$^?$

Consider a subset of eRPM, RPM$^?$, which does not contain any loops. In programs with conditional flows, the EDP of a region can not always be expressed as the completion of execution of just a single region. For example, if the EDP of some region r_i is expressed as r_j^+ and the region r_j resides within a conditional branch that may not be executed, then the deallocation point of r_i may be unspecified for some execution paths. This therefore breaks Step (1b) of the correctness criteria of EDP specifications. Just as with the multi-threaded versions of the RPM, the notation of how EDPs are specified must be extended and the memory requirements expression and correctness criteria derived.

Consider the program in Figure 4.9. The region r_2 has three possible deallocation points: r_5^+, r_7^+ and r_9^+. The chosen deallocation point depends on the result of the conditionals c_1 and c_2 that are carried out dynamically at runtime. The notation used to specify the EDP of a region r_i is $\text{EDP}^{\mathcal{M}^{\text{RPM}^?}}(r_i) = E$ where $E = \{T_1, T_2, \ldots, T_n\}$, each T_t is a tuple of the form $(\{C_{t,1}, C_{t,2}, \ldots, C_{t,m}\}, r_j^+)$ and each $C_{t,c}$ is a conditional region denoted c if the boolean result of executing the conditional region c is `true` and $\overline{c}$ if it is `false`. The size of E must be at least 1 and the number of conditions in each set in T_t can be zero.[14] For example, in Figure 4.9,

[14]If the size of E is 1 and no conditionals are present, then the region and its EDP lie on the same path with no conditionals between them. This is useful as it allows the same notation to describe programs in RPM$^{\rightarrow}$.

$\text{EDP}^{\mathcal{M}^{\text{RPM}^?}}(r_2) = \{(\{\overline{c_1}\}, r_5^+), (\{c_1, c_2\}, r_7^+), (\{c_1, \overline{c_2}\}, r_9^+)\}$. For simplicity, it is required that all conditionals in a track from r_i to r_j are listed. This can either be a requirement of how a specification is provided by a developer or could be automated. For example, if in Figure 4.9 it is specified that the deallocation point for (r_1) is always r_9^+ then this is equivalent to specifying $\text{EDP}^{\mathcal{M}^{\text{RPM}^?}}(r_1) = \{(\{c_1, c_2\}, r_7^+), (\{c_1, \overline{c_2}\}, r_9^+), (\{\overline{c_1}\}, r_5^+)\}$. Note that requiring EDPs to be specified in this way is important when checking for correctness and is therefore assumed to be the case.

Correctness of EDP Specifications in $\text{RPM}^?$

Let $\text{Tracks}(P)$ be the set of tracks in a program P. Let $\text{Tracks}(P, r_i)$ where $\text{Tracks}(P, r_i) \subseteq \text{Tracks}(P)$ be the set of tracks which contain the region r_i. A simple sanity check can be carried out first to ensure that an element of the set E identifies regions that are in the same track and therefore reachable from each other in a track. This means that when $\text{EDP}^{\mathcal{M}^{\text{RPM}^?}}(r_i) = \{T_1, T_2, \ldots, T_n\}$, then for every T_t in the form $(\{C_{t,1}, C_{t,2}, \ldots, C_{t,m}\}, r_j^+)$, there exists a track in $\text{Tracks}(P, r_i)$ which includes all conditional regions $\{C_{t,1}, C_{t,2}, \ldots, C_{t,m}\}$ as well as r_i and r_j.

Since the correctness criteria for EDPs described on page 79 require that every region has at least one deallocation point that is reached during the program's execution (Step (1a)) and of these deallocation points only one is guaranteed to be reached (Step (1b)), then these requirements are both satisfied if *every track* in $\text{Tracks}(P, r_i)$ has *exactly one* region whose execution completion can be identified as the deallocation point of r_i in that track. If a track were to have no deallocation point for a region, then Step (1a) would not be satisfied as the program could terminate without the memory for that region being deallocated. For example in Figure 4.9, if it is specified that $\text{EDP}^{\mathcal{M}^{\text{RPM}^?}}(r_1) = \{(\{c_1\}, r_4^+)\}$, then the specification is incorrect as there is no specified deallocation point for r_1 if executing c_1 returns `false`. On the other hand, if there were two deallocation points on a track, then, if the program were to follow this path, two deallocation points are guaranteed to be reached. Hence this fails to satisfy Step (1b). Requiring one deallocation point per track also ensures that no conflicts arise when considering the results of conditionals that could lead to ambiguity in identifying when a region is to be deallocated. For example, in Figure 4.9, if $\text{EDP}^{\mathcal{M}^{\text{RPM}^?}}(r_1) = \{(\{c_1\}, r_4^+), (\{c_1, \overline{c_2}\}, r_8^+), \ldots\}$, then the specification is incorrect as when executing r_1, the result of c_2 is unknown and therefore it is unclear whether the deallocation point will be r_4^+ or r_8^+.

Formally, if $\rho \varepsilon \text{Tracks}(P, r_i)$, then there *must* exist *exactly one* T_t in the EDP specification for r_i where T_t is in the form $(\{C_{t,1}, C_{t,2}, \ldots, C_{t,m}\}, r_j^+)$ for which all of $\{C_{t,1}, C_{t,2}, \ldots, C_{t,m}\}$ are in the regions in ρ. An implication of this requirement is that for every distinct pair of elements (C_i, r_i^+) and (C_j, r_j^+) in an EDP specification, C_i and C_j can not be subsets of each other. Note that using this approach, two tracks can share the same deallocation point. Moreover, it also possible to have

more than one deallocation point on the same execution path (but on a different track) due to different results of a conditional expression. This means that a memory subsystem must be able to identify at the deallocation point the path taken by the program. This approach subsumes the more simple one of requiring a single location to exist on the same path from r_{start} to r_{exit} and makes RPM$^?$ more expressive. Finally, in order to fulfill Step (1c) of the correctness criteria for an EDP specification, it must also be ensured that each deallocation point r_j^+ comes after r_i, that is $r_i \preceq r_j$ for every deallocation point r_j^+ in the pair of every T_t. This check also acts as a sanity check by ensuring no deallocation point for r_i falls at some location that can not be reached from r_i.

4.3.2 The Constrained RPM$^?$: cRPM$^?$

In cRPM$^?$, allocation and deallocation points of a region can be shifted in a similar way as cRPM$^\rightarrow$ is extended from RPM$^\rightarrow$. The constrained versions of the RPM models investigated so far all encompass their unconstrained counterparts as a special case. In this case, an expression for the memory requirements of cRPM$^?$ programs is derived immediately as RPM$^?$ is a special case of cRPM$^?$. The correctness criteria are extended first as this requires substantial changes to the way RDPs and RAPs are specified.

Specifying RAPs and RDPs in cRPM$^?$

The RAP for a region r_i in cRPM$^?$ is specified as a set $\text{RAP}^{\mathcal{M}^{\text{cRPM}^?}}(r_i) = A$ where A is a set of inter-region locations with elements in the form r_j^-. Unlike previous RPM models, RAPs and RDPs must be considered together in cRPM$^?$ as each element of an RDP specification depends on the allocation point specified in an RAP. For example, in Figure 4.9, if $\text{RAP}^{\mathcal{M}^{\text{cRPM}^?}}(r_4) = \{r_2^-\}$ then the conditional result of c_1 will determine whether r_4 is executed and a deallocation must be specified both if it does and if it does not. Therefore, the RAP of a region must be verified correct before the RDP is in turn checked for correctness. Moreover, each allocation point specified through an RAP must be paired with its corresponding set of deallocation points in the RDP specification.

The RDP of a region r_i, $\text{RDP}^{\mathcal{M}^{\text{cRPM}^?}}(r_i)$, is specified as a set of pairs where each element is in the form (α, β), α is a single element of the set $\text{RAP}^{\mathcal{M}^{\text{cRPM}^?}}(r_i)$ as defined above and β is a set of RDPs, each specified in the same way as EDPs in RPM$^?$. Every element in $\text{RAP}^{\mathcal{M}^{\text{cRPM}^?}}(r_i)$ must have exactly one corresponding (α, β) pair in $\text{RDP}^{\mathcal{M}^{\text{cRPM}^?}}(r_i)$.

Correctness of cRPM$^?$ Specifications

Step (2a) and (2b) of the correctness criteria for RAPs can be checked by requiring that for all tracks ρ in Tracks(P, r_i), there exists exactly one element r_j^- in $\text{RAP}^{\mathcal{M}^{\text{cRPM}^?}}(r_i)$ which is an element of the regions of ρ. Step (2c) is checked by requiring that for all allocation points r_j^- in $\text{RAP}^{\mathcal{M}^{\text{cRPM}^?}}(r_i)$, $r_j \preceq r_i$. This check also acts as a sanity check by ensuring no allocation point for r_i falls at some

location that r_i can not be reached. Note that with this approach to specifying RAPs, the path to an allocation point is irrelevant as all paths from the beginning of the program to that allocation point will have no other allocation points for r_i.

An RDP specification for a region r_i is correct if every (α, β) in that specification is correct as follows. Let $\alpha = r_j^-$ and B be an element of β so that $B = (\{C_{b,1}, C_{b,2}, \ldots, C_{b,m}\}, r_k^+)$. First, in order for the specification of the pair (α, β) to be correct, $\text{EDP}^{\mathcal{M}^{\text{RPM}^?}}(r_j) = \beta$ must also be correct in RPM$^?$. This ensures that each allocation point r_j^- has valid corresponding deallocation points in Tracks(P, r_j), thereby satisfying Steps (3a) and (3b) of the correctness criteria for RDPs. In order to satisfy Steps (3c) it is necessary to show that the deallocation points defined in every B in β do not identify an inter-region location that could occur before the deallocation point specified on that track is executed. Two cases need to be considered: r_i lying on the track identified by B and r_i not lying on the same track identified by B. For example, in Figure 4.9, if $\text{RAP}^{\mathcal{M}^{\text{cRPM}^?}}(r_7) = \{(\{\}, r_4^-)\}$ the corresponding deallocation points might be at r_7^+ or r_8^+ (the corresponding (α, β) pairs would be $((\{\}, r_4^-), (\{c2\}, r_7^+))$ and $((\{\}, r_4^-), (\{\overline{c2}\}, r_8^+))$ respectively) where r_7 does not lie on the track identified by $(\{\overline{c2}\}, r_8^+))$. If r_i lies on the track identified by B, then the EDP specification for r_i must also have a specified deallocation point along this track and if this point is r_s^+ then the specification of this pair is correct if $r_k^+ \geq r_s^+$ on this track. If r_i does not lie on the track identified by B, then the specified deallocation point must occur at a point in which it is guaranteed that r_i is not going to be executed, that is $r_k \prec r_i$ must not hold.[15] It is reiterated that as with all previous RPM models, an STS for program P is correct in cRPM$^?$ if all RAPs, EDPs and RDPs are correct for all regions in P.

Calculating Memory Requirements in cRPM$^?$

The memory requirements in programs with conditionals can be obtained by finding for each region the maximum memory requirements of that region and then taking the maximum of these values. The memory requirements for a region r_i are calculated by summing the sizes of all regions that could be deallocated after r_i and having been allocated before r_i. These regions are first identified using $\text{AllocatedAt}^{\mathcal{M}^{\text{cRPM}^?}}(r_i)$:

$$\begin{aligned}
\text{AllocatedAt}^{\mathcal{M}^{\text{cRPM}^?}}(r_i) \quad = \quad & \{r_x \ ; \\
& \exists\, \alpha \varepsilon \text{RAP}^{\mathcal{M}^{\text{cRPM}^?}}(r_x) \wedge \alpha = r_s^- \ \wedge r_s \preceq r_i \wedge \\
& \exists (\alpha, \beta) \varepsilon \text{RDP}^{\mathcal{M}^{\text{cRPM}^?}}(r_x) \wedge \exists\, B \varepsilon \beta \wedge B = (\cdot, r_j^+) \wedge \\
& \exists\, \rho = \langle R, E, \cdot, \cdot \rangle \varepsilon \text{Tracks}(P, r_s) \wedge r_i \varepsilon R \wedge r_j \varepsilon R \wedge \\
& r_j^+ > r_x\}
\end{aligned} \tag{4.22}$$

[15] Note however that $r_i \prec r_k$ could still hold.

The first line of the expression identifies those regions r_x that have an allocation r_s^- somewhere before r_i and on a track that passes through r_i. If at least one such track exists, then there is the possibility that r_x has memory for it allocated when executing r_i and therefore r_x could contribute to the memory requirements at the point when executing r_i. Having identified r_s^-, the deallocation points associated with this allocation point are the β values of this allocation point. For r_x to contribute to the memory requirements of r_i when considering r_s^-, at least one of these deallocation points must come after r_i is executed. Consider r_j^+ to be a deallocation point under consideration. The second and third lines therefore identify all the tracks in Tracks(P, r_s) and select the subset of these tracks that pass through both r_i and r_j as these are the only tracks of interest when considering the deallocation point r_j^+ .[16] If the allocation point under consideration (r_s^-) has an associated deallocation point r_j^+ on the tracks that includes r_i, then the fourth line checks whether r_j^+ comes after r_x. If it does, then r_x is added to the set AllocatedAt$^{\mathcal{M}^{\text{cRPM}^?}}(r_i)$. The memory requirements at r_i are then given by:

$$\text{MemReq}^{\mathcal{M}^{\text{cRPM}^?}}(r_i) = \sum\{\mid r_x \mid ; \; r_x \varepsilon \text{AllocatedAt}^{\mathcal{M}^{\text{cRPM}^?}}(r_i)\} \tag{4.23}$$

Using similar equations to Equations 4.7 and 4.8, it is now possible to find the memory requirements of the whole program.

Memory Sharing in RPM$^?$ **and** cRPM$^?$

The expression for calculating the memory requirements of programs in cRPM$^?$ as described above is used when memory allocated for a region can not be used by any other region. *Memory sharing*[17] for compile-time optimisation has been investigated in the past and has resulted in several techniques, a summary of which can be found in [59]. However, these approaches consider subsequences of the program for which memory is required for only the duration of that subsequence. Therefore, they are only relevant to the RPM for reducing the memory requirement of a single region through an analysis of memory sharing in intra-region flows. Existing work on offline allocation strategies on the other hand only considers straightline programs without the possibility of memory sharing [82, 94]. Although the work described by Gergov in [67] shows that the two problems are similar, there have been no attempts to reconcile these two pieces of information in order to find an offline allocation strategy that also takes advantage of memory sharing.

The decision of which regions in cRPM$^?$ to share memory for in order to derive an optimal solution that minimises the memory requirements of a program is NP-complete as both intra-region

[16] This is equivalent to finding the intersection of Tracks(P, r_s),Tracks(P, r_i) and Tracks(P, r_j).

[17] Memory sharing should not be confused with region sharing as described in Section 4.2.4 for RPM$^{\|\sigma\rho}$. Memory sharing involves using the same allocated memory to be the allocation context of more than one region. Region sharing is used to describe the objects in a region accessing objects in another region in another thread, thereby impacting the point at which deallocation of the memory for a region occurs.

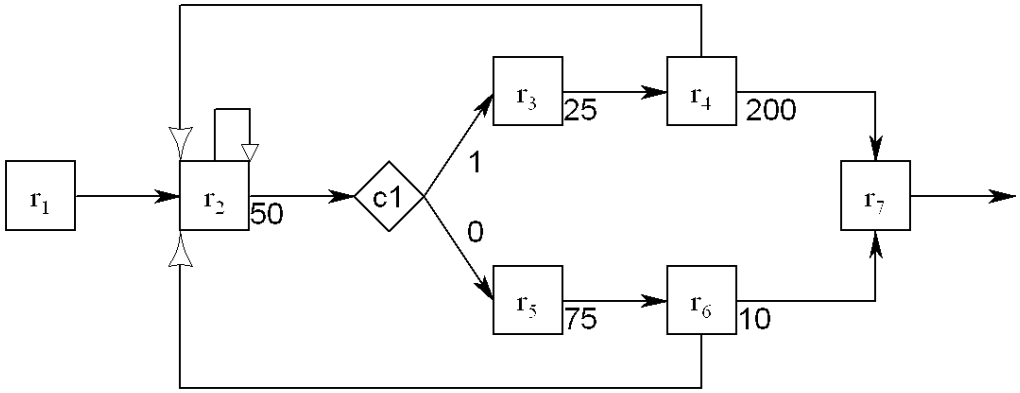


Figure 4.10: Sharing the Same Memory Between Regions

memory sharing and offline allocation based on lifetime are NP-complete. It is outside the scope of this thesis to investigate the two separate problems of finding a lower bound on the optimal memory requirements when taking advantage of memory sharing and known region lifetime, as well as polynomial algorithms that solve this problem. Instead, this section provides a brief investigation into why these problems are non-trivial and thereby describes a simple technique for implementing memory sharing in cRPM$^?$. It will be shown that this technique can reduce the memory requirements of a program when the allocation points are not all defined to occur just before the execution of regions. Therefore, this technique is only useful for programs in cRPM$^?$ as in RPM$^?$ the allocation point for a region r_i will always occur at r_i^-. Note that such an approach would require the memory subsystem to keep track of which regions are set to execute in the context of a given memory area.

A region r_i can use the memory allocated for a region r_j if and only if these two regions are not on the same track (that is $\neg(r_i \succ r_j \vee r_j \succ r_j)$). If an STS specifies such a set of regions, all of which have identical allocation points, then a single region can be used to be the allocation context of all these regions. The memory subsystem would therefore need to allocate a memory space equal to the maximum of the memory requirements of all these regions that can be used as the allocation context for any region of this set. The deallocation point of this shared region would be the latest point on every track when considering all the regions that would share this memory area. This is necessary to ensure that the contents of a region are not deallocated any earlier than specified in the RDP of that region.

The complexity of deciding whether to carry out this operation becomes immediately apparent for even this simplest of cases. Consider Figure 4.10[18] which shows a program in which the allocation points for r_4 and r_6 are both at r_2^-. Note that in this example the allocation point is the same for

[18]It is noted in passing that allocation points are depicted using the arrow types shown in this figure throughout this thesis.

all tracks in Tracks(P, r_4) and Tracks(P, r_6). In order to avoid the cumbersome notation of RDPs in cRPM$^?$, it is stated that the deallocation point for these α values are both r_4^+ and r_6^+ respectively if execution follows the path of these regions and c_1^+ if it does not. Therefore, if c_1 evaluates to `true`, then r_6 could be deallocated immediately. However, if c_1 evaluates to `false`, then r_4 could be deallocated immediately. The allocation points and deallocation points for all other regions come immediately before and after those regions respectively. The number besides each region r_i indicates $\mid r_i \mid$. All other regions have negligible memory requirements. Using Equation 4.23, the memory requirements are greatest at r_2 and are equal to $\mid r_2 \mid + \mid r_4 \mid + \mid r_6 \mid = 50 + 200 + 10 = 260$. The memory requirements at r_3 are $\mid r_3 \mid + \mid r_4 \mid = 25 + 200 = 225$ and those at r_5 are $\mid r_5 \mid + \mid r_6 \mid = 75 + 10 = 85$. However, if one memory area of size $\mid r_4 \mid$ is created for executing both r_4 and r_6 and assuming memory can not be resized, the memory requirements at r_5 are $\mid r_4 \mid + \mid r_5 \mid 200+75 = 275$. Note that if $\mid r_4 \mid$ and $\mid r_6 \mid$ were both 200 then this approach would have indeed reduced the memory requirements of the whole program to 275 ($\mid r_4/r_6 \mid + \mid r_5 \mid = 200 + 75 = 275$) from 450 ($\mid r_4 \mid + \mid r_6 \mid + \mid r_2 \mid = 200 + 200 + 50 = 450$).

If an STS sets an allocation point which is required only to observed by ensuring that a region is allocated by that point, then a further optimisation could be obtained by moving the allocation point even earlier to take advantage of the approach described above. For example, in Figure 4.10, if $\mid r_4 \mid$ and $\mid r_6 \mid$ were indeed both 200 and the STS set the allocation point for r_4 to r_1^-, then also moving the allocation point of r_6 to r_1^- would mean that this optimisation could be used. When moving allocation points, care must be taken to ensure the STS remains correct. In particular, if an allocation point of r_x is pushed back to that of r_y, $r_{y'}^-$, then all allocation points for r_x in each track Tracks(P, r'_y) must also be pushed back to $r_{y'}^-$.[19] The corresponding (α, β) pairs for r_x must then be updated. This requires first deleting all pairs for which the α value is any of the allocation points that have been moved. A new (α, β) is then added to the RDP of r_x where the α value is the new allocation point $r_{y'}^-$ and the β value is a set of deallocation points that is the latest of the deallocation points specified in the RDPs of r_x and r_y. The (α, β) pair for the allocation point $r_{y'}^-$ is also set to have this β value. A simple algorithm that reduces the memory requirements using this technique is sketched.

Using Equation 4.22, all the regions that contribute to the memory requirements at r_i can be found. Assume that r_i is the region which, when executing, incurs the largest memory requirements. If any of the regions in AllocatedAt$^{\mathcal{M}^{\text{cRPM}^?}}(r_i)$ are not on the same track, then the memory area in which they execute could be shared between them. By selecting two regions r_x and r_y in AllocatedAt$^{\mathcal{M}^{\text{cRPM}^?}}(r_i)$ (for example the two regions with the largest memory requirements) and moving one allocation point of that allocated later to the allocation point of the other allocated

[19] Note that because only one allocation point can exist on a track for a region, it is not possible that one allocation point of r_x comes before r_y and one comes after r_y.

earlier, the same memory can be used for executing both r_x and r_y and the memory requirements at r_i are reduced by the smallest of $\mid r_x \mid$ and $\mid r_y \mid$. The region with the smallest memory requirements is set size 0 and the deallocation points are shifted appropriately as described above. Clearly, this will impact the memory requirements of several other regions. In particular, those regions between r_i and the region set to 0 have an increase in memory requirements of size $\text{mod}(\mid r_x \mid - \mid r_y \mid)$ and the earlier allocation point will also impact the memory requirements of regions between the new allocation point and any old allocation points.

The algorithm then proceeds in two stages. First, if the memory requirements of the program are now reduced, then the algorithm is reapplied on the region which is now that with the greatest memory requirements. At the second stage, if the memory requirements are found to be greater, then the old allocation and deallocation points are restored and a new pair of regions in $\text{AllocatedAt}^{\mathcal{M}^{\text{cRPM}^?}}(r_i)$ are used in the algorithm. In order to provide an optimal solution, every possible pairing must be considered at the first stage of the algorithm before the second stage is applied and each pairing that gives a reduction in the memory requirements must be applied in the second stage. Due to the significant size of this search space, a tradeoff can be prescribed and the algorithm executed until a suitably low value for the memory requirements is found.

4.3.3 Iteration in the RPM: $\text{RPM}^{\circlearrowleft}$

A subset of eRPM, $\text{RPM}^{\circlearrowleft}$, which does not contain any conditional branches is considered next. $\text{RPM}^{\rightarrow}$ is extended to include the four types of loops: fixed loops, infinite loops, conditional loops and infinite&conditional loops. Recall that fixed loops are executed for a pre-defined number of times, infinite loops, as their name implies, are never exited whereas conditional loops are executed as long as executing a conditional region returns `true` and a pre-defined iteration limit is not reached. This predefined iteration limit could be set to ∞ making that loop an infinite&conditional loop. The way EDPs are specified, first for fixed loops and infinite loops and then conditional loops and infinite&conditional loops, is described next. This is followed by an exploration of the correctness criteria for programs in $\text{RPM}^{\circlearrowleft}$. The specification of RAPs and RDPs in $\text{cRPM}^{\circlearrowleft}$, the constrained version of $\text{RPM}^{\circlearrowleft}$, is described next. Finally, the correctness criteria and memory requirements of programs in $\text{cRPM}^{\circlearrowleft}$, are derived.

Specifying EDPs in Programs With Fixed and Infinite Loops Only

Consider first a subset of $\text{RPM}^{\circlearrowleft}$ which does not contain any conditional loops. This model shall remain unnamed as it serves only as a means to provide a gradual progression to the full $\text{RPM}^{\circlearrowleft}$. Therefore, EDPs will be specified without the $\mathcal{M}^{\text{RPM}^{\circlearrowleft}}$ superscript to avoid confusion with the complete $\text{RPM}^{\circlearrowleft}$ that will eventually be developed.

Consider the program in Figure 4.11 with regions $r_{\overrightarrow{1,6}}$. The program consists of 6 regions with

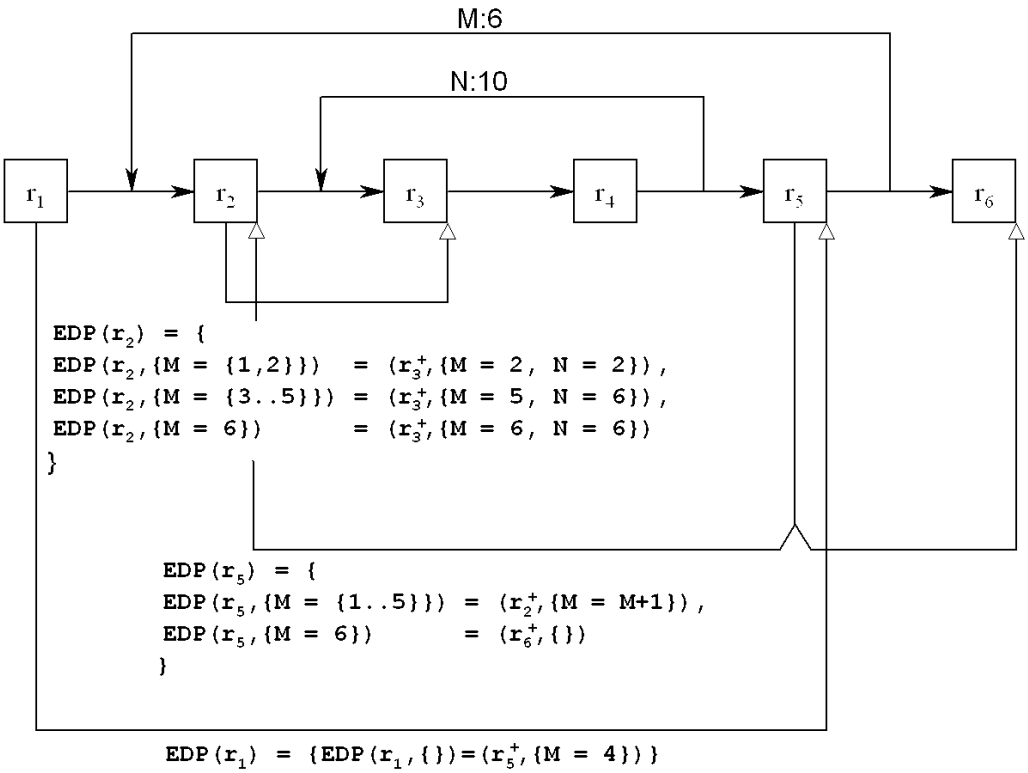


Figure 4.11: An Example of a Program with Fixed Loops

two fixed loops M and N that execute for 6 and 10 iterations respectively. The diagram shows how the EDPs of regions are specified when they and their deallocation points fall within a loop. Region r_1 does not reside within a loop but, as its deallocation point does, the EDP specification must specify which execution completion of r_5 in the loop M is this point. This is true of any region with a deallocation occurring within a loop. If the deallocation point falls within a nested loop, then this must be identified with a set of loop identifiers. Iterated regions can have a unique deallocation point for every iteration of that region. For example, in Figure 4.11 the region r_5 is executed six times in the loop M and a deallocation point needs to be specified for every execution of this region in its EDP. The notation used to specify EDPs is therefore $\text{EDP}(r_i) = E$ where E is a set with elements of the form $\text{EDP}(r_i, \{L_1, L_2, \ldots, L_n\}) = (r_j^+, \{G_1, G_2, \ldots, G_m\})$. The elements $L_1, \ldots L_n$ identify which executions of r_i the deallocation point is valid for. A region nested in n loops would have n of these elements. Each L_x takes the form $L = l$ where L is the loop identifier and l is the iteration number. Two shorthand notations can be used whereby L is either a list of integers (for example $L = \{1, 3, 5\}$) or a range of integers (for example $L = \{1..3\}$). A combination of lists and ranges is also possible. The elements $G_1, \ldots, G_m$ identify which execution completion of r_j is the deallocation point. The elements of this set have the form $G = g$ where G is a loop identifier and g is some integer specifying which iteration of G the expression holds for. Once again, a region nested in n loops would have n of these elements. If some loop L_x in $\{L_1, \ldots, L_n\}$ is the same loop as some loop G_y in $\{G_1, \ldots, G_m\}$ then the iteration for the latter loop can be expressed in terms of the former. An example of this is shown for the loop M in the definition of the EDP for r_5 in Figure 4.11. Again, this is simply a convenient shorthand notation. For the purposes of deriving the correctness criteria and memory requirements of programs, shorthand notations are expanded to their full form.

The deallocation points of regions executing in infinite loops must be described in a finite manner in the EDP of these regions. In order to achieve this, an expression is used to identify the iterations for which an element of E in the EDP holds. For every region r_i in the infinite loop, a finite number of executions of r_i can have their deallocation points specified as above for fixed loops. All other regions must use an expression pair to specify their deallocation points. The notation used is $\text{EDP}(r_i) = E$ where E is a set with each element in the form $\text{EDP}(r_i, A \bigcup \{\textit{InequalityExpr}\} \bigcup B) = (r_j^+, X \bigcup \{\textit{Expr}\} \bigcup Y)$ where A, B, X and Y are sets of pairs of loop identifier and corresponding iteration number and *InequalityExpr* and *Expr* are an equality expression and inequality expression expressed in terms of the infinite loop I. For example, if $\text{EDP}(r_i, \{I > 0\}) = (r_j^+, \{I = I + 3\}) \varepsilon \text{EDP}(r_i)$, then the deallocation point of the I^{th} execution of r_i in the loop would be the completion of execution of the $I^{th} + 3$ iteration of r_j. It is assumed that regions in an infinite loop would have deallocation points that are specified to lie within the infinite

loop. When this is not the case, then this would imply that the memory created by these regions is never freed. Therefore, for every region r_i in the infinite loop, unless $\mid r_i \mid = 0$ or, if $\mid r_i \mid > 0$, an infinite number of these region executions fail to respect this format, then the program will need infinite memory. Also, assuming that all deallocation points of regions in an infinite loop fall within that loop, and as execution control never exits the infinite loop, it can be assumed that an infinite loop would be the outermost loop and A and X above are empty. Hereafter, it is assumed that this is the case; a program where this is not the case can easily be transformed by stripping the outer loops.[20] Moreover, as regions after the infinite loop can never be executed, these regions can be removed from the program and any deallocation points specified as the execution completion of these regions will be allocated for the lifetime of the application. In summary, in a program with one or more infinite loops in RPM$^{\circlearrowleft}$, the first and most deeply nested infinite loop I is identified, all outer loops of I are removed and all regions after the last region of I are also removed. Any region executed before the infinite loop can have its deallocation point specified as the completion of execution of a region in I. Also, any region with a deallocation point after any of the deleted regions will never be deallocated. A tool implementing this minimisation would need to notify the user of deleted loops and regions as in all probability this would occur due to an error in the STS.

Specifying EDPs in Programs With Conditional and Infinite&Conditional Loops Only

As in programs with fixed and infinite loops, regions in conditional loops must have a specified deallocation for every possible iteration of that loop, even if some iterations may not be executed. In order to satisfy correctness requirements of EDP specifications in RPM$^{\circlearrowleft}$, regions with deallocation points that fall within a conditional loop must have alternative deallocation points specified in the event that the inter-region location in the conditional loop is not reached. The EDP of region r_i is therefore specified as $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\circlearrowleft}}}(r_i) = E$ where E is a set with elements of the form $\text{EDP}(r_i, \{L_1, L_2, \ldots, L_n\}) = D$ and D is a set with elements in the form $(r_j^+, \{G_1, G_2, \ldots, G_m\})$. Note that the superscript for the EDP specification of r_i is now set as $\mathcal{M}^{\text{RPM}^{\circlearrowleft}}$ as this notation is more general than that for fixed and infinite loops given above and will therefore be the notation used to specify allocation and deallocation points in the entire RPM$^{\circlearrowleft}$ that considers all of fixed, infinite, conditional and infinite&conditional loops. However, the superscript $\mathcal{M}^{\text{RPM}^{\circlearrowleft}}$ is suppressed for the elements in E as it is understood that these are relevant to this model from the notation $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\circlearrowleft}}}(r_i)$. As with the notation for fixed loops, the elements L_i and G_i identify which executions of r_i and r_j the deallocation point holds for. An example of this notation is given in Figure 4.12.[21] Here, region r_1 has a deallocation point that occurs either after the execution of the specified iterations of r_4 and r_5 in loops $c1$ and $c2$ or after r_6 is executed, whichever is reached first.

[20] Note that this will not also be true of infinite&conditional loops.

[21] The superscript $\mathcal{M}^{\text{RPM}^{\circlearrowleft}}$ for $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\circlearrowleft}}}(r_1)$ is omitted.

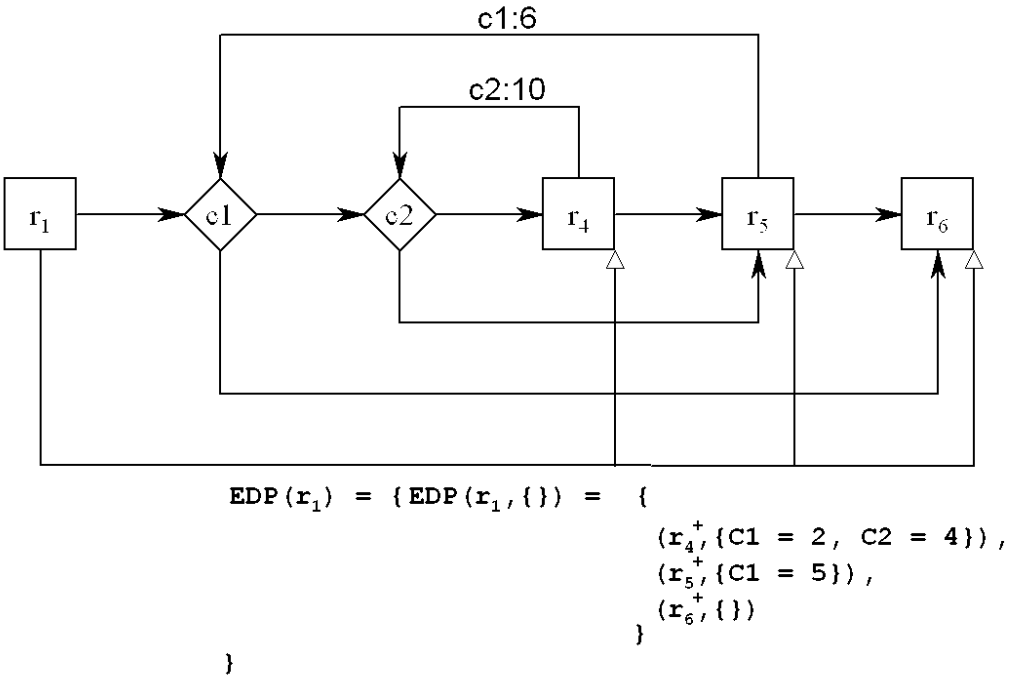


Figure 4.12: An Example of a Program with Conditional Loops

A key issue pertaining to allocation and deallocation inter-region locations must be explained in further detail before deriving the correctness criteria programs with conditional loops. When a conditional loop C is reached, the first inter-region location to be reached is $(c^-, X \bigcup \{C = 1\})$ where c is the conditional region of the loop C which is nested in loops X. If c evaluates to `true`, then the regions in the conditional loop are executed. Once this iteration is complete, $(c^-, X \bigcup \{C = 2\})$ is reached. At the point that the conditional region c evaluates to `false`, it is considered that $\overline{c}$ has been executed and the next inter-region location is the inter-region location for specifying deallocation points $(\overline{c}^+, X)$. Therefore, $(c^-, X \bigcup \{C = 1\})$ is a guaranteed allocation point if the conditional loop is reached, as is $(\overline{c}^+, X)$. Also, the allocation point $(c^-, X \bigcup \{C = i+1\})$ is always guaranteed to be reached from any region in the conditional loop C when executing the i^{th} iteration of C.

Recall that when programs in RPM$^{\circlearrowleft}$ with no conditional loops were considered, a single deallocation point was specified for every iteration of a region in an infinite loop by using an equality and inequality pair. It was also argued that this deallocation point had to occur within the infinite loop if the memory requirements of that region were non-zero. With the introduction of conditional regions in infinite&conditional loop, several deallocation points can be specified. If the outer loop is a fixed infinite loop however, it remains the case that all these deallocation points must fall within this infinite loop. The correctness criteria for RPM$^{\circlearrowleft}$ developed below will consider this further. In contrast

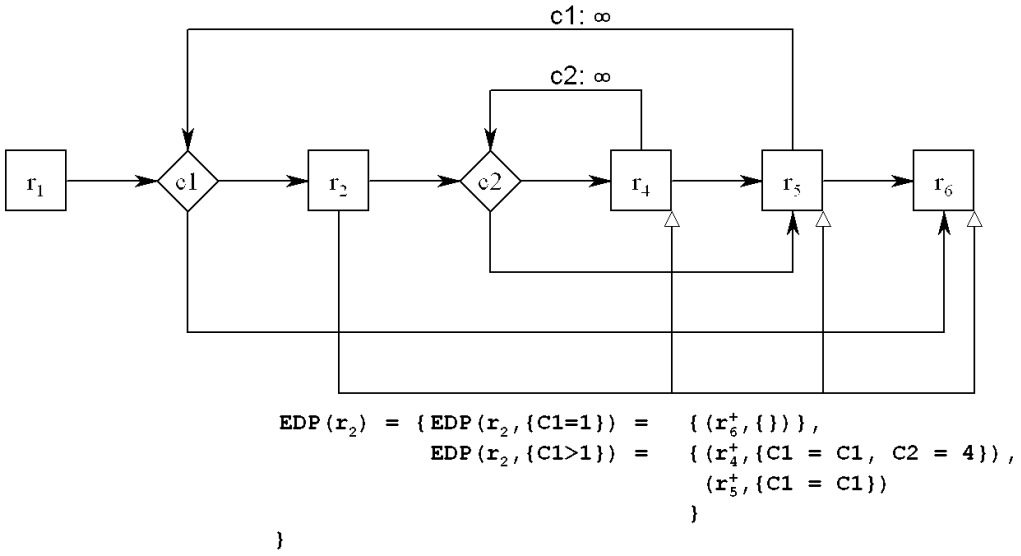


Figure 4.13: An Example of a Program with Infinite Conditional Loops

to infinite loops however, infinite&conditional loops may be nested and there can be other regions after them. Moreover, although some deallocation points for a region in an infinite&conditional loop will have to be specified within that same loop, some deallocation points will be able to fall after that loop. Again, this will be explained in more detail when defining the correctness criteria for RPM$^{\circlearrowleft}$. It suffices here to define the notation used to express EDPs for regions in RPM$^{\circlearrowleft}$ that fall within either type of infinite loop. A region r_i in possibly nested infinite loops has an EDP specified as $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\circlearrowleft}}}(r_i) = E$ where each element in E is of the form $\text{EDP}(r_i, \mathit{InequalityExprs} \bigcup X) = D$. *InequalityExprs* is a set of inequality expressions for the nested infinite loops (one of which may be a infinite loop and not an infinite&conditional loop) and X is a set of fixed or conditional loops.[22] Similarly, D is a set of elements of the form $(r_j^+, \mathit{Exprs} \bigcup Y\})$ where *Exprs* is a set of equality expressions for the infinite loops and Y is a set of fixed or conditional loops. Note that for some elements of D, *Exprs* can be empty since some deallocation points may fall outside infinite, conditional loops. An example of the notation used is given in Figure 4.13. Here, the deallocation point of region r_2 in the first iteration of the conditional loop with conditional region $c1$ is at r_6^+. If this conditional loop never exits, then $(r_2, \{c1 = 1\})$ is never deallocated. All other iterations of r_2 are deallocated either at r_4^+ in the fourth iteration of this conditional loop or, if this region is not executed, at r_5^+ in the same iteration of the loop.

[22]Note that whereas the order at which loops were specified previously was in the order of nesting, this is not the case here.

Specifying EDPs in Programs in RPM$^{\circlearrowleft}$

EDPs in programs with conditional loops are specified in a more general way to programs with fixed and infinite loops. Therefore, the notation used to describe EDPs in programs in RPM$^{\circlearrowleft}$ is the same as that for programs with conditional loops described above.

Correctness of EDP Specifications for Programs in RPM$^{\circlearrowleft}$

Consider first the subset of RPM$^{\circlearrowleft}$ that contains no infinite loops or infinite&conditional loops. Recall that for a conditional loop with conditional region c, if the conditional loop is reached, then c will always execute one more time than the number of times the loop is iterated. Also, $\overline{c}^{+}$ is the only deallocation point that is guaranteed to be executed. Since Step (1b) of the correctness criteria for EDPs requires that exactly one deallocation point is guaranteed to be reached for every possible region execution, these two observations must be considered when ensuring this condition.

The first thing to note is that any deallocation points that fall within a conditional loop with conditional region c (except for $\overline{c}^{+}$) must be assumed to never be executed. Therefore (with one exception), satisfying Step (1b) of the correctness criteria for EDPs described on page 79 requires every execution of a region to have exactly one deallocation point that falls outside a conditional loop. The only exception to this rule would be when both the region and its deallocation point fall within the same iteration of the conditional loop. This is because the region whose execution completion marks the deallocation point will be guaranteed to execute. An additional sanity check in addition to the correctness criteria is specified that requires the single guaranteed deallocation point shown to exist in Step (1b) to come after any deallocation points that are not guaranteed to be reached. The reason that the deallocation point that is guaranteed to be reached can not come before those that are not is that if this were to occur, then the latter would be redundant. Correctness of EDPs specifications for programs in this subset of RPM$^{\circlearrowleft}$ requires six checks as follows:

1. Step (1a) of the correctness criteria for EDPs is satisfied when, for an EDP specification $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\circlearrowleft}}}(r_i) = E$ for a region r_i in nested loops $L_1, L_2, \ldots, L_n$, the size of the set E is equal to the product of the number of iterations of $L_1, L_2 \ldots, L_n$. This ensures that the number of times a region is executed is the same as the number of sets of deallocation points specified for that region. Note that conditional loops are considered as fixed loops executing for the maximum number of iterations at this step but with an extra iteration of the conditional region of the loop for the iteration of the loop.

2. The second check ensures that Step (1c) of the correctness criteria for EDPs is satisfied; that is that every specified deallocation point of an execution of r_i is an inter-region location that will be reached after that iteration of r_i. Note that this does not yet check the problem of

conditionals described above. If $\text{EDP}(r_i, X) = D$ is in $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\circlearrowleft}}}(r_i)$, $(r_j^+, Y)\varepsilon D$ and no loop identifier in X can be found in Y, then the EDP specification is incorrect if $r_i > r_j$. If X and Y have loop identifers in common, then the EDP specification might still be correct if $r_i > r_j$ as the deallocation point might refer to the execution of r_j in a subsequent iteration. This somewhat complicates the specification of correctness as it can no longer be defined on $<$ alone. Therefore, an expression is defined so that a region execution (r_j, Y) is executed after (r_i, X) if $\text{Succeeds}((r_i, X), (r_j, Y))$. An implementation of $\text{Succeeds}((r_i, X), (r_j, Y))$ is achieved by comparing the iteration numbers of the loops common to both X and Y and the order of these two regions under $<$. Note that by definition, a region also succeeds itself. This is necessary to allow for a deallocation point of a region that immediately follows that region's execution. For an EDP specification to be correct, it must therefore be shown that for every element $\text{EDP}(r_i, X), = D$ in $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\circlearrowleft}}}(r_i)$ and every element (r_j^+, Y) in D, $\text{Succeeds}((r_i, X), (r_j, Y))$. At this point, it is ensured that every iteration of a region has at least one deallocation point and all these points come after the execution of that region, therefore satisfying Step (1c). Next it is necessary to ensure that exactly one deallocation point is guaranteed to execute as required in Step (1b). As described in the next two steps, this is achieved by first showing that each region has a non-empty set of deallocation points that are guaranteed to be reached and then showing that the size of this set is 1.

3. To ensure that each region has at least one deallocation point that is guaranteed to be executed it must be shown that for all $\text{EDP}(r_i, X) = D\varepsilon\text{EDP}^{\mathcal{M}^{\text{RPM}^{\circlearrowleft}}}(r_i)$, one of the following must hold:

   (a) there must be exactly one element in D of the form (r_j^+, Y) where Y contains no conditional loops. If the outermost region of Y is the conditional loop C for which c is its conditional region, $(\overline{c}^+, \{C = 1, \ldots\}$ is not considered to reside in the loop and is therefore an element of this set.

   (b) If the set from (3a) is empty, then there must be exactly one element in D for which every conditional loop in Y also appears in X and for which the iteration numbers of these conditional loops are identical. Note that in this case r_i is itself in a conditional loop. Once again, the conditional region c for any conditional loop C at this iteration has a guaranteed deallocation point at $\overline{c}^+$ and is therefore an element of this set.

4. It is noted that Step (3) only ensures that at least one deallocation point is reached. In order to guarantee that there is exactly one deallocation point that is guaranteed to be reached, it is necessary to reconsider Step (3a) and Step (3b). If Step (3a) holds and r_i is not in a conditional loop, then this requirement is satisfied as only (r_j, Y) is guaranteed to execute and it is the only region outside a conditional loop. However, if r_i is in a set of nested conditional

loops then all regions in the same iterations of these conditional loops are also guaranteed to execute. Therefore, if (3a) holds and there is an element of D in the form $(r_{j'}^{+}, Y')$ where all conditional loops in Y' are in X and at the same iteration number, (that is (3b) also holds) then two guaranteed deallocation points have been identified and the specification is incorrect. It therefore follows that for there to be exactly one deallocation point specified, *only one* of (3a) and (3b) can hold.

5. The sanity check described above to ensure that the single guaranteed deallocation point found in the two previous steps comes after all other deallocation points requires that if (r_j, Y) is the single guaranteed deallocation point, then Succeeds$((r_{j'}, Y'), (r_j, Y))$ must be true for all other deallocation points $(r_{j'}, Y')$.

6. Finally, another sanity check required to remove redundant deallocation points can be ensured. If a non-guaranteed deallocation point is reached, then no other deallocation point in the set of non-guaranteed deallocation points is guaranteed to already have been reached at this point. For example, if a region outside a conditional loop has a deallocation point in a conditional loop at the i^{th} iteration of this loop, then this check would require that there is no other deallocation point at this or any previous iteration of this loop which is not itself further nested in another conditional loop.

Note that in order to ensure EDPs are correct, other simple sanity checks could be carried out. These might include ensuring that loop identifiers are valid in a context and that iteration numbers specified as part of an EDP are indeed within the fixed loop limits specified in the program. It is assumed that such pre-validation is carried out.

Correctness of EDPs in programs with infinite and infinite&conditional loops in addition to fixed and conditional loops is considered next. Consider a region r_i in possibly nested fixed loops, conditional loops, an infinite loop and infinite&conditional loops that has an EDP specified as $\text{EDP}^{\mathcal{M}^{\text{RPM}^{\circlearrowleft}}}(r_i) = E$ where each element in E is of the form $\text{EDP}(r_i, \mathit{InequalityExprs} \bigcup X) = D$. D is a set of elements of the form $(r_j^{+}, \mathit{Exprs} \bigcup Y\})$. Recall that X and Y hold loop identifiers for fixed and conditional loops whereas *InequalityExprs* and *Exprs* describe the loop iterations for the possible infinite loop and the infinite&conditional loops in the program. Recall also that if an infinite loop exists in the program, then it will be the only such loop and it will also be the outermost loop, nesting within it any combination of fixed, conditional and infinite&conditional loops. Step (1b) of the correctness criteria for an EDP specification requires that it is shown that every execution of a region has exactly one deallocation point that is guaranteed to execute in the future. This means that it must be proven that if the set *Source* is an infinite set containing every execution of a region r_i in the infinite loops and *Dest* is a similar set that contains only regions *guaranteed* to

be executed for r_i, then a function can be defined that maps every element of *Source* to *Dest*. This function need not be surjective or injective but it must be a function (ie not a multivalue or partial function) with a domain that includes every element of *Source* that can be mapped to exactly one element of *Dest*. Similar checks to those in Steps (5) and (6) above can be carried out. Such a proof requires knowledge of the theory of infinite numbers [71] and is beyond the scope of this thesis. Satisfying Step (1a) of the correctness criteria for EDP specifications can be shown if the set *Source* encompasses every possible iteration of r_i. In order to satisfy Step (1c), it must also be shown that in every possible pairing in this function, the element in *Source* is executed before every element in *Dest*. Once again such a proof may be non-trivial. However, if it can be shown that the values of the loop iterations in *Source* are always smaller than those in *Dest*, then this would be a sufficient proof. Finally, a sanity check similar to that employed for programs without infinite loops can be required to ensure that all non-guaranteed deallocation points for a given region execution come before the deallocation point specified in the set *Dest* for that region execution.

As an example of proving the correctness of programs with infinite loops and infinite&conditional loops, consider a program which has a region r_1 that is nested inside an infinite&conditional loop C and an infinite loop I and a region r_2 that is nested in the same infinite loop I as r_1. Let $\mathrm{EDP}(r_1) = E$ where $E = \{EDP(r_1, \{I > 0, C > 0\}) = D\}$ and $D = \{(r_1^+, \{I = I, C = C + 1\}), (r_2^+, \{I = I\})\}$. The correctness of this EDP specification is investigated. Step (1a) of the correctness criteria for EDPs requires that every execution of r_1 has a set of deallocation points specified. Since the single element in E encompasses all possible executions of r_1 ($r_1, \{I > 0, C > 0\}$ is specified), then this condition is satisfied. r_1 has two possible deallocation points every time it is executed: at r_1^+ in the subsequent iteration of the conditional loop or, if this loop iteration is not executed, at r_2^+ of the same iteration in the loop I. Since none of the first set of inter-region locations are guaranteed to be reached, the set *Dest* contains only the inter-region locations in the second set. As every execution of r_i has exactly one of these locations specified as a deallocation point, then Step (1b) of the correctness criteria for EDPs is satisfied. Finally, since the deallocation points in both sets of inter-region locations for a region execution of r_i are reached after that execution of r_i, then Step (1c) of the correctness criteria for EDPs is also satisfied. The sanity check that ensure that guaranteed deallocation points come after non-guaranteed ones also holds in this case since the inter-region locations in the second set are always reached after those in the first.

4.3.4 The constrained RPM$^{\circlearrowleft}$: cRPM$^{\circlearrowleft}$

RPM$^{\circlearrowleft}$ is extended to describe programs in cRPM$^{\circlearrowleft}$ that consider early allocation and delayed deallocation. The correctness criteria for RAP and RDP specifications are derived together with an expression for the memory requirements of cRPM$^{\circlearrowleft}$ programs.

Specifying RAPs and RDPs in Programs in cRPM$^{\circlearrowleft}$

Similar to deallocation points in EDP specifications in RPM$^{\circlearrowleft}$, each region in cRPM$^{\circlearrowleft}$ can have more that one allocation point specified in its RAP that depends on the execution path taken. Therefore, for a region r_i, the RAP of a r_i is specified in a similar way to EDPs; that is, $\text{RAP}^{\mathcal{M}^{\text{cRPM}^{\circlearrowleft}}}(r_i) = A$ where A is a set with elements of the form $\text{RAP}(r_i, \{L_1, L_2, \ldots, L_n\}) = D$[23] and D is a set with elements in the form $(r_j^-, \{G_1, G_2, \ldots, G_m\})$. An example of this notation used in describing the RAPs of r_4 and r_6 in Figure 4.14 would be:

$$
\begin{aligned}
\text{RAP}^{\mathcal{M}^{\text{cRPM}^{\circlearrowleft}}}(r_4) = \quad & \{ \\
& \text{RAP}(r_4, \{M = \{1 \ldots 6\}, c1 = 1\}) = \{(r_2^-, \{\})\} \\
& \text{RAP}(r_4, \{M = \{1 \ldots 3\}, c1 > 1\}) = \{(r_3^-, \{M = M, c1 = c1\})\} \\
& \} \\
\\
\text{RAP}^{\mathcal{M}^{\text{cRPM}^{\circlearrowleft}}}(r_6) = \quad & \{ \\
& \text{RAP}(r_6, \{M = \{1 \ldots 3\}\}) = \{(r_4^-, \{M = M, c1 = 3\}), (r_5^-, \{M = M\})\} \\
& \text{RAP}(r_6, \{M = \{4 \ldots 6\}\}) = \{(r_4^-, \{M = M, c1 = 4\}), (r_5^-, \{M = M\})\} \\
& \}
\end{aligned}
\tag{4.24}
$$

In this example the memory for the first iterations of r_4 in the conditional loop are all allocated at r_2^- (six in all) and that for all other iterations of this loop is allocated at r_3^- at the same iteration of the conditional loop. The memory for the first three iterations of r_6 is allocated in the conditional loop at r_4^- and at the third iteration of this conditional loop and in the same iteration of the fixed loop. If the conditional loop does not reach its third iteration, then the memory for these regions is allocated before r_5 is executed in the same iteration of the fixed loop. Similarly, the memory for the last three iterations of r_6 is allocated in the conditional loop at r_4^- at the fourth iteration of the fixed loop and the fourth iteration of the conditional loop. If the fourth iteration of the conditional loop is not reached, then the memory for these last three iterations of r_6 is also allocated before r_5 is executed and at the same iteration of the fixed loop.

Deallocation points for programs in cRPM$^{\circlearrowleft}$ are specified in a similar way to EDPs but every deallocation point specified for a region iteration must be paired with the allocation point for which it holds. Therefore, for some region execution $\text{RAP}(r_i, X) = A \varepsilon \text{RAP}^{\mathcal{M}^{\text{cRPM}^{\circlearrowleft}}}(r_i)$, let α be an element

[23] Note that the superscript $\mathcal{M}^{\text{cRPM}^{\circlearrowleft}}$ is suppressed for the elements in A as it is understood that these are relevant to this model from the notation $\text{RAP}^{\mathcal{M}^{\text{cRPM}^{\circlearrowleft}}}(r_i)$.

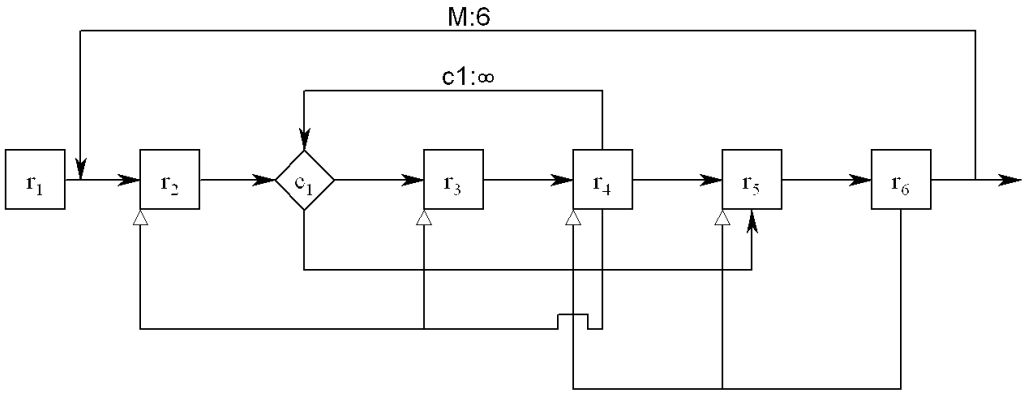


Figure 4.14: Specifying RAPs in cRPM$^{\circlearrowleft}$

of A. RDP$(r_i, X) = D$ will be an element of RDP$^{\mathcal{M}^{\text{cRPM}^{\circlearrowleft}}}(r_i)$ and the pair (α, β) will be an element of D. β is then a set of deallocation points specified in the same way as EDPs. Therefore, every allocation point α for a region iteration is paired with the set of possible deallocation points that will be valid if the inter-region location α is reached. Allowing the existence of both non-guaranteed and guaranteed allocation points means that the memory subsystem must keep a record of where every conditional loop exited that loop so that when a deallocation point for a region is reached, a decision can be taken on whether the deallocation point holds for the path taken.

Correctness of RAP and RDP Specifications for Programs in cRPM$^{\circlearrowleft}$

There are two approaches that could be adopted for checking correctness of RAPs and RDPs in cRPM$^{\circlearrowleft}$. The first approach is to carry out a loop unrolling operation of the cRPM$^{\circlearrowleft}$ program and assign the allocation and deallocation points of the RAP and RDP specifications as defined for cRPM$^{\circlearrowleft}$ into RAPs and RDPs for cRPM$^{?}$. The correctness criteria for cRPM$^{?}$ can then be used to check the correctness of the original specifications. This approach is investigated shortly when describing how the memory requirements for programs in cRPM$^{\circlearrowleft}$ are found. The second approach is to consider the RAP and RDP specifications directly.

Consider first a subset of cRPM$^{\circlearrowleft}$ which contains no infinite loops. The correctness of RAPs in cRPM$^{\circlearrowleft}$ follows a similar process as that for EDPs and RDPs. Every region iteration must have exactly one guaranteed allocation point and this must come after any non-guaranteed ones but before the execution of that region. Recall that if a conditional loop is reached, then the inter-region allocation point before its conditional region and at the first iteration is guaranteed to execute. Also, the $i^{th} + 1$ inter-region allocation point of the conditional region of the loop is guaranteed to be reached if the i^{th} iteration of the conditional region evaluates to `true`. The five rules similar to the six described for EDPs are:

1. Step (2a) of the correctness criteria for RAPs is satisfied when, for an RAP specification $\text{RAP}^{\mathcal{M}^{\text{cRPM}^{\circlearrowleft}}}(r_i) = A$ for a region r_i in nested loops $L_1, L_2, \ldots, L_n$, the size of the set A is equal to the product of the number of iterations of $L_1, L_2 \ldots, L_n$. This ensures that the number of times a region is executed is the same as the number of sets of allocation points specified for that region. Note that an extra iteration of the conditional region of conditional loops must be taken into account at this step.

2. The second check ensures that Step (2c) of the correctness criteria for RAPs is satisfied; that is that every specified allocation point of an execution of r_i is an inter-region location that will be reached before that iteration of r_i. For an RAP specification to be correct, it must therefore be shown that for every element $(r_i, X), = D$ in $\text{RAP}^{\mathcal{M}^{\text{cRPM}^{\circlearrowleft}}}(r_i)$ and every element (r_j^-, Y) in D, Succeeds$((r_j, Y), (r_i, X))$.

3. To ensure that each iteration of a region has exactly one allocation point that is guaranteed to be executed as required by Step (2c) of the correctness criteria for RAPs, it must be shown that for all $\text{RAP}(r_i, X) = D \varepsilon \text{RAP}^{\mathcal{M}^{\text{cRPM}^{\circlearrowleft}}}(r_i)$, exactly one of the following must hold:

   (a) there must be exactly one element in D of the form (r_j^-, Y) where Y contains no conditional loops.

   (b) there must be exactly one element in D of the form (r_j^-, Y) where r_j is the conditional in the outermost conditional loop in Y and this conditional loop C is specified as $C = 1$ in Y.

   (c) If there is no element in D of the form (r_j^-, Y) for which Y contains no conditional loops, then there must be an element (r_k^-, Z) in D for which every conditional loop in Y also appears in X and for which Succeeds$((r_k, Z), (r_i, X))$.[24] Note that, similar to programs in cRPM$^?$, it is possible for a region to have memory for it allocated but that region is then itself never executed. Nevertheless, the correctness criteria for RDPs guarantee that this memory is deallocated at some point.

4. The sanity check to ensure that the single guaranteed allocation point found in the previous steps comes after all other allocation points requires that if (r_j, Y) is the single guaranteed allocation point, then Succeeds$((r_{j'}, Y'), (r_j, Y))$ must be true for all other allocation points $(r_{j'}, Y')$.

[24] Note that there is a small difference here to Step (3a) on page 112 for checking correctness of EDPs in RPM$^{\circlearrowleft}$. In the case of EDPs, there was no guarantee that subsequent iterations of a conditional loop would execute. However, with RAPs, if a region execution is reached for a set of nested conditionals, then it is guaranteed that all previous iterations have also been executed.

5. Finally, if a non-guaranteed allocation point is reached, then no other allocation point in the set of non-guaranteed allocation points is guaranteed to already have been reached at this point.

The correctness of an RDP specification for a region r_i requires that every iteration of r_i, (r_i, X) has (α, β) pairs such that $\text{EDP}(\alpha) = \beta$ is correct when using the correctness criteria for iterations of region in RPM$^{\circlearrowleft}$. This ensures that Steps (3a) and (3b) of the correctness criteria for RPDs are satisfied. Secondly, Step (3c) of these criteria requires that all deallocation points specified in the RDP must come after the first deallocation point specified in the EDP even though it could be argued that deallocation points in the RDP must all come after the single guaranteed deallocation point in the EDP. The reason the former approach is chosen is because it makes logical sense that if a specification identifies a point as a deallocation point in an EDP, then there exists an implied guarantee that the memory required for that region can be freed anywhere after that point. Let (r_j^+, Y) be the earliest executed region in D, where $\text{EDP}(r_i, X) = D$ is an element of $\text{EDP}^{\mathcal{M}^{\text{cRPM}^{\circlearrowleft}}}(r_i)$. If $\text{RDP}^{\mathcal{M}^{\text{cRPM}^{\circlearrowleft}}}(r_i) = E'$ and $\text{RDP}(r_i, X) = (\alpha, \beta)$ is an element of E' then for every element $(r_{j'}^+, Y')$ in β, $\text{Succeeds}((r_j, Y), (r_{j'}, Y'))$ must hold.

The RDPs and RAPs of regions inside infinite loops and infinite&conditional loops require a similar treatment as EDPs. A proof based on the values of the expressions for regions in the loop would have to be derived using a similar approach to that for EDPs in RPM$^{\circlearrowleft}$.

Calculating Memory Requirements in cRPM$^{\circlearrowleft}$ with no Infinite Loops

The memory requirements of individual regions in previously described versions of the RPM such as cRPM$^{\rightarrow}$ and cRPM$^{?}$ were derived by identifying for each region r_i, those regions R in the program that could have memory allocated for them while r_i was executing. Since RPM$^{\circlearrowleft}$ and cRPM$^{\circlearrowleft}$ allow regions to be executed more than once, they are subject to a different set R at each iteration. In order to calculate the worst case memory requirements of r_i, it is therefore necessary to identify which iteration of r_i is subject to the greatest memory requirements. Consider first the subset of cRPM$^{\circlearrowleft}$ that contains no infinite loops. Since every iteration of a region can have a unique set of allocation and deallocation points specified in the RAP and RDP of that region, finding the memory requirements for each region can be achieved by "unrolling" the loop. There are two techniques that can be adopted to carry out this process. The first involves converting cRPM$^{\circlearrowleft}$ programs to cRPM$^{\rightarrow}$ programs, thereby allowing the equations for cRPM$^{\rightarrow}$ to be used to calculate memory requirements. The second involves converting cRPM$^{\circlearrowleft}$ programs to cRPM$^{?}$ programs, thereby allowing the equations for cRPM$^{?}$ to be used instead. In carrying out either of these operations, it is necessary to show how the allocation points and deallocation points specified in RAPs and RDPs as defined for cRPM$^{\circlearrowleft}$ programs are transformed to RAPs and RDPs in cRPM$^{\rightarrow}$ or

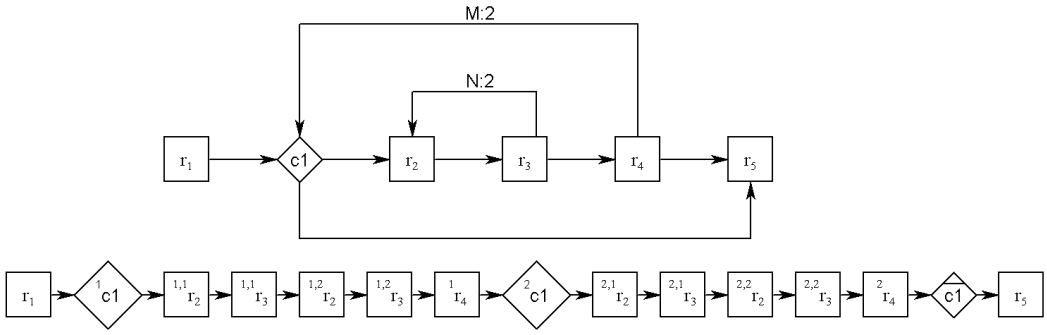


Figure 4.15: Loop Unrolling of Programs in cRPM$^{\circlearrowleft}$ to Programs in cRPM$^{\rightarrow}$

cRPM$^{?}$. It must also be shown that the derived RAPs and RDPs obtained in this transforation are correct in cRPM$^{\rightarrow}$ or cRPM$^{?}$ if RAPs and RDPs were also correct in cRPM$^{?}$. These two techniques are considered in turn.

Unrolling of loops to convert cRPM$^{\circlearrowleft}$ programs to cRPM$^{\rightarrow}$ programs is achieved by copying the contents of that loop for the number of iterations that loop is executed. Therefore, if the BNF description of a fixed loop is `(n,<BlockList>)`, then this is converted to `(<BlockList>,<BlockListRest>)`, where `<BlockListRest>` is a `<LoopBlock>` entity in the form `(n-1,<BlockList>)`. This is repeated recursively until $n = 1$ in which case `(n,<BlockList>)` is converted to `(<BlockList>)`. Conditional loops are treated as fixed loops for this purpose but with an extra region added at the end to represent the execution of the condition of the loop executing the final time and returning `false`. An example of this unrolling operation is shown in Figure 4.15. Note that the notation used for describing a region iteration is just an alterative for the current notation used (that is, (r_i, X)).

In calculating the worst case memory requirements of a region r_i, the two points that must be considered for r_i are the earliest possible allocation point (which may not be the single guaranteed allocation point) and the single guaranteed deallocation point (which will be the last deallocation point that can be reached). This gives the longest possible lifetime of the memory allocated for r_i and, therefore, $\mid r_i \mid$ is added to the memory requirements of the largest number of regions. Therefore, if for some region execution (r_i, X), the first allocation point of (r_i, X) is (r_h^-, W) and the last deallocation point is (r_j^+, Y), then $\mathrm{RAP}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_i, X) = (r_h^-, W)$ and $\mathrm{RDP}^{\mathcal{M}^{\mathrm{cRPM}^{\rightarrow}}}(r_i, X) = (r_j^+, Y)$.[25]

The correctness of the derived specification is trivial in this approach. If there is indeed only one guaranteed deallocation point in cRPM$^{\circlearrowleft}$, then for a given region iteration, exactly one allocation point and deallocation point exist in the cRPM$^{\rightarrow}$ specification. If two guaranteed allocation points exist in the cRPM$^{\circlearrowleft}$ for a given region iteration, then two guaranteed deallocation points would appear in the RDP specification for that region in cRPM$^{\rightarrow}$ and the specification is therefore incorrect.

[25]Note that regions (r_h, W), (r_i, X) and (r_j, Y) are just an encoding of the unrolled program similar to that used in Figure 4.15.

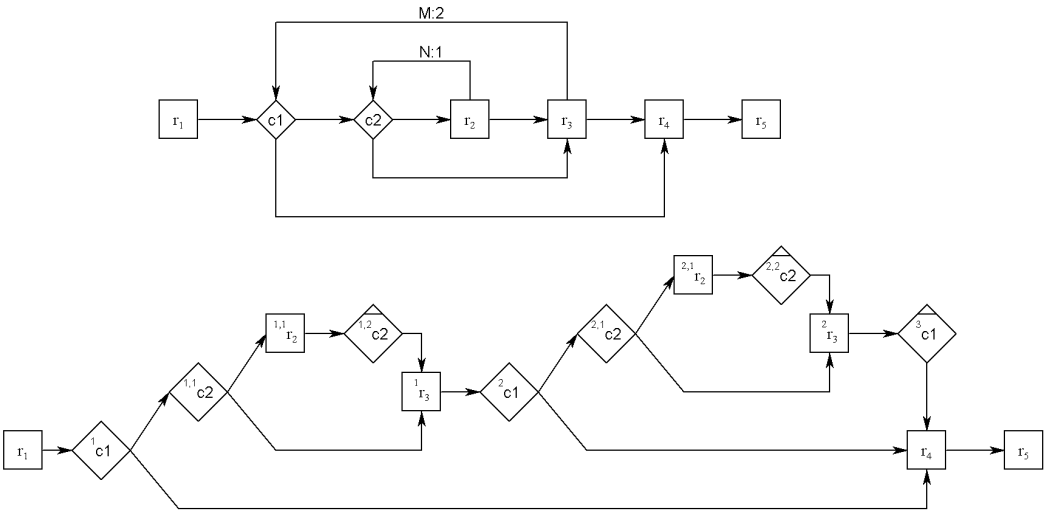


Figure 4.16: Loop Unrolling of Programs in cRPM$^{\circlearrowleft}$ to Programs in cRPM$^{?}$

Note that the converse is not true: the transformation could lead to a specification that is correct in cRPM$^{\rightarrow}$ that was not originally correct in cRPM$^{?}$. This is because there could be two guaranteed allocation point in the RAP specification in cRPM$^{\circlearrowleft}$. Therefore, it would be necessary to check correctness of the program in cRPM$^{\circlearrowleft}$ before this transformation is carried out.

Considering only the guaranteed deallocation point of a region execution can be shown to be pessimistic in calculating the memory requirements using this approach. This is because a non-guaranteed deallocation point of a region execution (r_i, X) could guarantee that another region execution (r_j, Y) does not incur the memory requirements of (r_i, X), even if (r_j, Y) is on the path between r_i and the guaranteed deallocation point. For example, if $\text{RDP}^{\mathcal{M}^{\text{cRPM}^{\circlearrowleft}}}(r_1) = \{\text{RDP}(r_1, \{\}) = \{(r_2^+, \{M = 1, N = 2\}), (r_5^+, \{\})\}\}$ in Figure 4.15, then all regions executed between $(r_3, \{M = 1, N = 2\})$ and $(r_4, \{M = 2, N = 2\})$ (denoted ${}^{1,2}r_3$ and ${}^{2,2}r_4$ in Figure 4.15) inclusive need not incur the memory requirements of $(r_1, \{\})$ since the memory for r_1 would be guaranteed to be deallocated at the point the program reaches these regions. One solution to address this is to first calculate the memory requirements of a region execution (r_i, X) using the first allocation point and last deallocation point of every other region iteration in the program as described above. The memory requirements of those regions iterations that are guaranteed to be deallocated when executing (r_i, X) are then subtracted from this value. Identifying these region iterations for (r_i, X) can be achieved by a simple operation: if some region iteration (r_j, Y) contributes to the memory requirements of (r_i, X) and there is a deallocation point specified for (r_j, Y) from which (r_i, X) is guaranteed to be reached, then $\mid r_j \mid$ can be subtracted from the memory requirements of (r_i, X).

Unrolling of loops to convert cRPM$^{\circlearrowleft}$ programs to cRPM$^{?}$ programs is achieved by listing every possible path the program can take from start to termination. Although this approach would

clearly require greater space requirements in an implementation, it has the advantage that both guaranteed and non-guaranteed deallocation points are considered immediately. Therefore, memory requirements can be calculated immediately once RAPs and RDPs are defined and a proof could be devised for a transformation to show that the derived program in cRPM$^?$ is correct if and only if the original program in cRPM$^{\circlearrowleft}$ was also correct. In this approach, fixed loops are unrolled in the same way as above. Conditional loops on the other hand are expanded so that if the BNF description of the conditional loop is `(c,n,<BlockList>)`, this is converted to a `<CondBlock>` entity in the form `(c,<BlockListRest>,null)` where `<BlockListRest>` is equal to `(<BlockList>,<BlockListRestOfLoop>)` and `<BlockListRestOfLoop>` is `(c,n-1,<BlockList>)`. This is repeated recursively until $n = 2$ inclusive. Finally, the last iteration of the loop is unrolled in a similar way but with an additional region to represent the execution of c returning `false` and exiting at the last iteration. Therefore `(c,1,<BlockList>)` is converted to `(c,(<BlockList>,c),null)`. An example of this unrolling operation is shown in Figure 4.16.

It must next be shown how RAPs and RDPs in cRPM$^{\circlearrowleft}$ are transformed into RAPs and RDPs in cRPM$^?$. The flexibility that has been provided to allow non-guaranteed and guaranteed allocation points becomes a problem here as every non-guaranteed allocation points and the single guaranteed allocation point will lie on the same track. Had allocations points in cRPM$^?$ been sensitive to the path taken, then every non-guaranteed allocation points could have been mapped on a single track with the guaranteed allocation point being used for all other tracks in Tracks$(P, (r_i, X))$.[26] As this is not the case, a suitable alternative would be to place all allocation points in the unrolled program and require that every track in this program has at least one allocation point (rather than exactly one). Therefore, if an inter-region location (r_j^-, Y) is an allocation point of some region execution (r_i, X) (that is (r_j^-, Y) is an element of RAP(r_i, X) in RAP$^{\mathcal{M}^{\text{cRPM}^{\circlearrowleft}}}(r_i)$), then all tracks in Tracks$(P, (r_i, X))$ in the unrolled program that have region (r_j^-, Y) as one of the regions on that tracks have an allocation point at (r_j^-, Y) (that is (r_j^-, Y) is an element of RAP$^{\mathcal{M}^{\text{cRPM}^?}}(r_i, X)$). At runtime, the memory subsystem would therefore have to mark regions for which memory has already been allocated in order that they will not be allocated again when another allocation point is reached.

Recall that deallocation points in RDPs in cRPM$^?$ and cRPM$^{\circlearrowleft}$ must be paired with one of the allocation points specified in the RAP of a region. Also, unlike RAPs, the deallocation point is sensitive to the path taken through the program. Therefore, every allocation point expanded as shown above has its own set of deallocation point. These deallocation points are obtained by first considering the set of non-guaranteed deallocation points specified in the unrolled cRPM$^{\circlearrowleft}$

[26] This would in fact only be possible if no other allocation point in the set of non-guaranteed allocation points is guaranteed to already have been reached at this point. This is the sanity check specified at Step 5 for checking the correctness of RAP specifications in cRPM$^{\circlearrowleft}$.

program in relation to that allocation point. Assuming fixed loops are already unrolled, if a non-guaranteed deallocation point is specified as $(r_j^+, \{C1 = c1, C2 = c2, \ldots, Cn = cn\})$ for some region execution (r_i, X), then (C, r_j^+) is a deallocation point in β value in the RDP specification of (r_i, X) in cRPM$^?$. C is a set of conditionals which all evaluate to `true` and for which every loop iteration of Ci has an element in C. For example, if a deallocation point is specified as $(r_j^+, \{C1 = 3\})$ for some region execution (r_i, X), then $(\{^1C1, ^2C1, ^3C1\}, r_j^+)$ is a deallocation point in the RDP specification of (r_i, X) in cRPM$^?$. For all tracks in the unrolled program for which a deallocation point is not specified by this transformation, the deallocation point on that track is specified to be the single guaranteed deallocation point. Therefore, if no other non-guaranteed deallocation point is guaranteed to have been reached by this point,[27] then there is one deallocation point for every track in Tracks$(P, (r_i, X))$ which is paired with every possible allocation point specified in the RAP. Therefore, Steps (3a) and (3b) of the correctness criteria for RDP specifications are satisfied. Step (3c) must also hold as all the deallocation points specified in an RDP specification must come at earliest as the first deallocation point specified in the EDP specification. Since the order of region execution is not changed in the unlooping operation, this constraint still holds.

Calculating Memory Requirements in cRPM$^\circlearrowleft$ with Infinite Loops

Infinite and infinite&conditional loops can not be "unrolled" in the same way as fixed loops. Consider first regions that fall in an infinite loop but not infinite&conditional loops only and consider that only EDPs are specified (ie the program is in RPM$^\circlearrowleft$ rather than cRPM$^\circlearrowleft$. Recall that the allocation and deallocation points in infinite loops will always fall within that loop when the STS is minimised. Also the single-element subset of these points that is guaranteed to execute will not fall in any conditional loop other than if the deallocation point is at the same iteration as the region and iteration number under consideration. The memory requirements for the loop must be guaranteed to reach a stable finite point after a number of iterations. Again, this requires a proof specific to the STS, a simple example of which is investigated next. Consider a program P in RPM$^\circlearrowleft$ which consists of two regions r_1 and r_2 executing in an infinite loop I and with no further nesting. Let $\text{EDP}^{\mathcal{M}^{\text{RPM}^\circlearrowleft}}(r_1) = \{EDP(r_1, \{I > 0\}) = \{(r_2^+, \{I = I{+}1\})\}\}$ and $\text{EDP}^{\mathcal{M}^{\text{RPM}^\circlearrowleft}}(r_2) = \{EDP(r_2, \{I > 0\}) = \{(r_2^+, \{I = I\})\}\}$. Therefore, the memory for each iteration of r_1 is deallocated after r_2 is executed in the subsequent iteration of the loop I. The memory for every iteration of r_2 is deallocated immediately after that iteration of r_2 is executed. By considering which regions are allocated at a particular point, it is trivial to show that the memory requirements of P at the first iteration of I are equal to $\mid r_1 \mid + \mid r_2 \mid$ and for subsequent iterations are equal to $2 \mid r_1 \mid + \mid r_2 \mid$. The memory requirements therefore reach equilibrium at this point. If, however, $\text{EDP}^{\mathcal{M}^{\text{RPM}^\circlearrowleft}}(r_1) = \{EDP(r_1, \{I > 0\}) = \{(r_2^+, \{I = I^2\})\}\}$, the memory requirements can be shown to be infinite as

[27] This is the sanity check specified at Step 6 for checking the correctness of EDP specifications in RPM$^\circlearrowleft$.

the memory requirements at every iteration i of I are equal to $(i + \lceil\sqrt{i}\rceil - 1) \mid r_1 \mid + \mid r_2 \mid$. This is because an iteration i of region r_1 can only be deallocated at iteration i^2. Note that proving that at some point the memory requirements of a program reach an equilibrium state does not mean that the memory requirements will become constant. Depending on the complexity of the inequality and equality expressions for I, the memory requirement could be shown to be finite but could still be seen to change as I tends to infinity. Such complexity would in practice be rare. Note also that the above example assumes that the allocation point for each iteration of r_1 and r_2 falls at the LAP of these regions and the deallocation point is specified by the EDP. An equivalent argument can be made for programs in cRPM$^{\circlearrowleft}$ with allocation and deallocation points specified by RAPs and RDPs. Infinite conditional loops are treated in a similar way to fixed loops but with the required proof having the added complexity of multiple inequality and equality expression pairs.

4.3.5 Combining Loops and Conditionals: eRPM

All the unconstrained versions of the RPM investigated so far allowed the developer to express as much information possible about the lifetime of regions. This was achieved by allowing every iteration of a region to have a unique deallocation point as specified in the EDP of that region. In turn, the constrained versions of the RPM also allowed as much information to be expressed so that any memory model that placed constraints on the allocation and deallocation points could do this as exactly as possible. Not only was each execution of a region allowed a unique set of allocation and deallocation points, but the deallocation point could also depend on the allocation point that was executed at runtime.[28] Such flexibility however comes at a cost, not only in an algorithm that implements the checks for correctness and the equations for memory requirements but also in runtime costs. For example, the pairing of allocation and deallocation points meant that the runtime had to recall the path taken through the program in order to only apply relevant deallocation points.

In eRPM, the combination of conditionals and loops begins to prove a significant burden on the runtime if the same approach of allowing as much information to be expressed as possible is adopted. If a conditional branch lies within a loop, then unrolling that loop would exponentially increase the number of possible tracks. For example, if a program consisted of a single fixed loop of m iterations and contains n conditional branches within it, then 2^{nm} possible tracks would exist in the program. It is possible to allow the specification of an allocation or deallocation point for every possible path through the loop, thereby relying on loop unrolling to check for correctness and calculate memory requirements as done in cRPM$^{\circlearrowleft}$. However, this approach would have two disadvantages for programs in eRPM. Firstly, the runtime overheads could be severely increased as the result of each iteration of a conditional region would have to be recalled by the memory subsystem. Secondly this would

[28] Indeed, even further flexibility could have been achieved by allowing allocation points to depend on the path taken by the program.

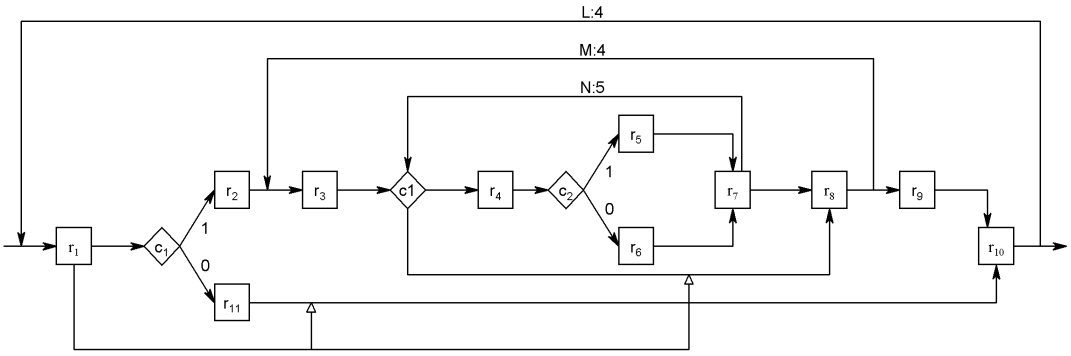


Figure 4.17: Specifying EDPs in eRPM$^{\maltese}$

make the notation of RAP and RDPs excessively cumbersome, allowing the flexibility of expressing lifetime information of regions that will probably not be useful to a memory model. For instance, it will rarely be the case that a memory model could benefit from the ability of having a unique allocation for every iteration of a region that also depends on the path taken till that point and which, furthermore, also benefits from a unique deallocation point for every allocation point.

Subsets of eRPM and indeed any version of the RPM described in this chapter can be developed to allow the expression of region lifetime as necessary for a class of application. In every case, the development process for these models would follow the same approach as taken here: a description of how EDPs, RAPs and RDPs in that model are specified is provided first followed by a definition of the correctness criteria for these specifications and an expression for the memory requirements of programs in this model. For completeness, a subset of a full eRPM model called eRPM$^{\maltese}$ is considered next. eRPM$^{\maltese}$ constrains the flexibility of how region lifetimes are specified in comparison to previous models in order to provide a simpler way of expressing deallocation points in EDPs. A brief investigation into the correctness criteria and memory requirements of programs in eRPM$^{\maltese}$ is then carried out.

Specifying EDPs, RDPs and RAPs in eRPM$^{\maltese}$

EDPs in eRPM$^{\maltese}$ are specified in the same way as for EDPs in cRPM$^{\circlearrowleft}$ but for which every region iteration has a deallocation point that follows a pattern based on the current iteration number if that deallocation point falls in the same loop. For example, in the program in Figure 4.17, the EDP of r_1 is specified as $\text{EDP}^{\mathcal{M}^{\text{eRPM}}}(r_1) = \{\text{EDP}(r_1, L = \{1..4\}) = \{(r_{11}^{+}, \{L = L\}), (\overline{c1}^{+}, \{M = 1, L = L\})\}\}$. Therefore, the size of the set $\text{EDP}^{\mathcal{M}^{\text{eRPM}^{\maltese}}}(r_1)$ is always equal to one. Note that should it be necessary for some iteration to have a different EDP, loop unrolling can always be employed. By using a pattern of allocation and deallocation for a region, the burden on the runtime is reduced as only one inter-region location in each case needs to be recalled for every region. If the deallocation

point for a region in that loop were to fall outside that loop, then it could be required that all regions are deallocated at the same point. Another approach to simplifying specifications is that the path taken through the program is irrelevant to which deallocation point is executed. RAPs and RDPs would be described in a similar fashion. In addition, RDPs do not have to depend on which allocation point was hit, thereby further reducing the amount of information that needs to be recalled in the runtime. This is equivalent to saying that the (α, β) pairs are made up of the each allocation point in the RAP paired with every deallocation point in the EDP.

Correctness of EDP, RAP and RDP Specifications for Programs in eRPM$^{\maltese}$

In order to check the correctness of EDP, RAP and RDP specifications in eRPM$^{\maltese}$, a loop unrolling operation is carried in order to derive a program in cRPM$^{?}$. The loop unrolling operation is identical to that carried out for cRPM$^{\circlearrowleft}$ programs. Note that conditional branches will be replicated in this case. An example of this operation is given in Figure 4.18. In order to check that the EDP for a region r_i is correct ($\text{EDP}^{\mathcal{M}^{\text{eRPM}^{\maltese}}}(r_i) = E$), then for all $(\text{EDP}(r_i, X) = D)\varepsilon E$ and all elements $(r_j^+, Y)\varepsilon D$ a deallocation point is placed at (r_j^+, Y) in the unrolled program for region (r_i, X). Once, again, several deallocation points would be found on the same track meaning that the correctness criteria using cRPM$^{?}$ would be weakened so that at least one deallocation point would have to be found on every track, rather than exactly one. Therefore, although the runtime would not have to recall the path taken to that track, it must recall whether memory for a region has already been deallocated or not. A similar approach is carried out in order to derive the RAP and RDP specifications in cRPM$^{?}$ and the same correctness criteria can be adopted whereby at least one allocation point rather than exactly one allocation point would have to be found on every track in $\text{Tracks}(P, (r_i, X))$. Although not proven here, it is trivial to show that the loop unrolling operation will create a program in cRPM$^{?}$ with all possible paths through the program. Therefore, if the original program in eRPM$^{\maltese}$ had exactly one allocation and deallocation point, then there will be at least one allocation point and deallocation point for $\text{Tracks}(P, (r_i, X))$ in cRPM$^{?}$ since these points would have to be in the regions of all of $\text{Tracks}(P, (r_i, X))$.

Calculating Memory Requirements in eRPM$^{\maltese}$

Having derived the RAP and RDP specifications in cRPM$^{?}$ from those in eRPM$^{\maltese}$, Equation 4.23 can be used to find the memory requirements of every region iteration. The memory requirements of the whole program are then the maximum of all these values. Note that Equation 4.23 remains valid despite the possibility of multiple allocation and deallocation points on the same track as it does not require a program specification correct in cRPM$^{?}$. However, because of the existence of patterns in the lifetime of regions, it may not be necessary to unroll all loops if it can be shown that memory requirements of regions also follow some pattern. This is of use in particular when

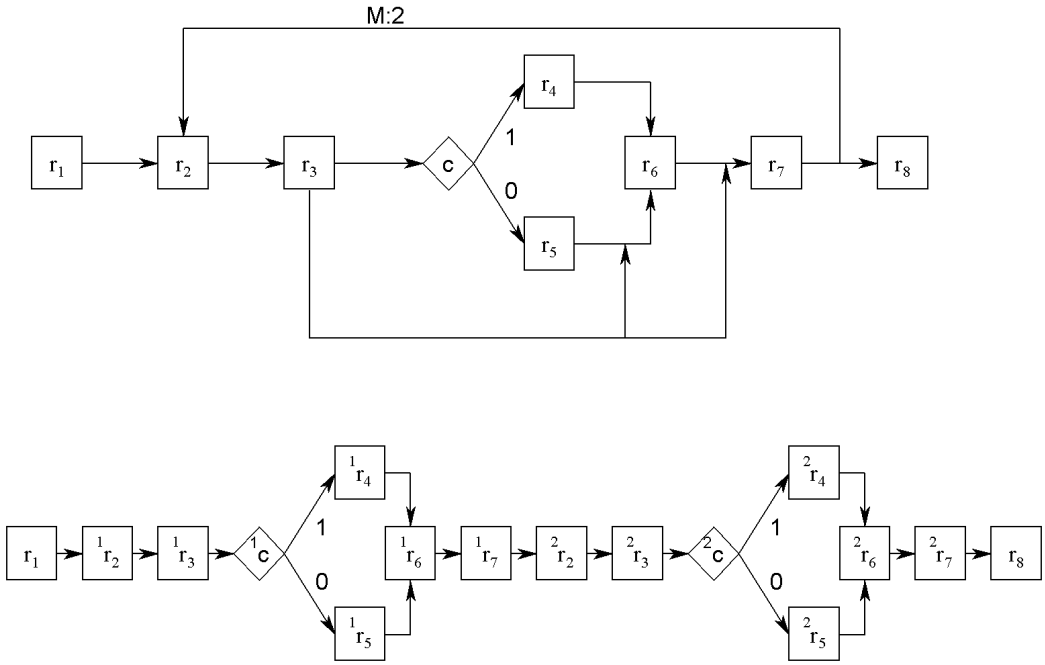


Figure 4.18: Loop Unrolling in eRPM

calculating the memory requirements of programs with loops with a large number of iterations as well as infinite loops. Moreover, annotations that mark RAPs and RDPs can be themselves annotated with information describing this pattern so that the overhead of retaining this information at runtime is reduced. An example of this technique is given in Section 5.5 where the annotations for a scoped model carry information about the lifetime of several iterations of the same region.

4.3.6 Multi-Threading in eRPM: eRPM$^{\|\sigma\rho}$ and eRPM$^{\|}$

In Section 4.2, multi-threading was introduced whereby every thread was specified as a RPM$^{\rightarrow}$ or cRPM$^{\rightarrow}$ program. Allowing multi-threading without region sharing and synchronisation is a trivial extension similar to that described in Section 4.2.1 for cRPM$^{\|}$. This model will be denoted as eRPM$^{\|}$. It is also possible to introduce multi-threading to allow individual threads to contain iteration and selection as in eRPM while still considering region sharing as in RPM$^{\|\sigma}$ and synchronisation as in RPM$^{\|\sigma\rho}$. As the different versions of the RPM described in this chapter suffice to prove the three parts of the thesis hypothesis, the model will not be extended further here. Appendix C describes an initial investigation into how programs in eRPM$^{\|\sigma\rho}$ might be described in order to express region sharing and synchronisation. The work carried out here shows that this extension is not trivial and requires special consideration of operations such as region sharing in programs with infinite loops in order to guaranteed finite memory requirements. Operations such as partitioning of threads in order to derive execution and memory traces becomes a significant problem which

requires further research.

4.4 Applying the RPM to RPM − Unscoped

In Section 3.4, a memory model was described as a combination of a Lifetime Specification Language (LSL) and a memory subsystem. As a simple example, a memory model that uses one of the RPM models as its LSL and one of two memory subsystems based on unscoped explicit allocation and deallocation is introduced. RPM − Unscoped is a general name used to describe a complete memory model that uses a memory subsystem that does not follow a scoped order. In RPM − Unscoped, the developer specifies only a set of EDPs for each region execution in the program. All regions are therefore allocated at their LAP and deallocated at the location specified in the EDP (that is the RDP specifications in $\mathfrak{D}$ are the same as the EDPs in $\mathfrak{E}$). The location in memory allocated for each region depends on the chosen memory subsystem. Two memory subsystems are described here: EMM − ONLINE, an online memory subsystem that does not take advantage of information on the lifetime of regions, and EMM − OFFLINE, a memory subsystem that uses offline analysis to derive an allocation strategy. For example, if $\text{RPM}^{\rightarrow}$ is to be used as a LSL, and EMM − ONLINE as the memory subsystem, then $\text{RPM} - \text{Unscoped} = \langle \text{RPM} - \text{Unscoped}^{\text{RPM}^{\rightarrow}}, \text{RPM} - \text{Unscoped}_{\text{EMM}-\text{ONLINE}} \rangle$. The shorthand notation for this tuple is $\text{RPM} - \text{Unscoped}^{\text{RPM}^{\rightarrow}}_{\text{EMM}-\text{ONLINE}}$.

4.4.1 The EMM − ONLINE Memory Subsystem

EMM − ONLINE is not itself a DMA in the sense used in Chapter 2. Rather, it acts as a middleware layer, enforcing the semantics of the RPM and passing allocation and deallocation calls down to any arbitrary DMA. As its name implies, EMM − ONLINE uses a DMA that operates in the spirit of *malloc()*/*free()*-style explicit allocation and deallocation. An implementation of EMM − ONLINE for a program in $\text{RPM}^{\parallel}$ without synchronisation or region sharing is trivial. The runtime first creates the immortal memory region in which static objects can be shared. Next, the required number of threads are created and the sequence of regions in each thread are executed. An implementation of EMM − ONLINE for $\text{eRPM}^{\parallel}$ would adopt a similar approach but execution of each thread would follow the control flow graph. In either case, every time a new iteration of a region r_i is about to be executed, the memory subsystem EMM − ONLINE issues a *malloc(*$\mid (r_i, X) \mid$*)* command to the DMA. When a region (r_i, X) finishes executing, EMM − ONLINE issues a *free(*(r_i, X)*)* command. Therefore, given a program specification, the program can be annotated with *malloc()* and *free()* calls which are used to mark the lifetime boundaries of regions and which can then be interpreted by EMM − ONLINE.

Region synchronisation for $\text{RPM}^{\parallel\sigma\rho}$ can be implemented using a rendezvous policy either in

the runtime or as part of EMM − ONLINE. Region sharing requires that deallocation points for a region are coordinated so that memory is not deallocated too early. A simple implementation of this requirement would be to have a bitmap of deallocation points for each region execution where a bit specifies whether the deallocation point for that region has been reached in one of the threads in the program. For each region execution, the size of this bitmap would be equal to the number of threads in the program. Therefore, rather than annotating programs with *free()* calls at deallocation points, a call to a function in EMM − ONLINE is made that sets the bit for the current thread. EMM − ONLINE then calls *free()* only when all bits have been set.

The memory requirements given by $\text{MemReq}^{\text{RPM}}(P)$ would give the memory requirements of the program using the appropriate equations for the chosen RPM model without the additional overheads of the memory subsystem EMM − ONLINE. $\text{MemReq}_{\text{EMM-ONLINE}}(P)$ which would have to be added to $\text{MemReq}^{\text{RPM}}(P)$ to give the total memory requirements $\text{MemReq}^{\text{RPM}}_{\text{EMM-ONLINE}}(P)$. Three additional overheads of EMM − ONLINE and its DMA must be considered: the overheads of any information stored by EMM − ONLINE; the overheads of the book-keeping structures of the DMA; and the extra space required due to fragmentation. The space overheads of EMM − ONLINE would include information such as the STS and the bitmap patterns for regions as described above and would be minimal in relation to $\text{MemReq}^{\text{RPM}}(P)$. What is more important is to understand what effect the approach taken by using EMM − ONLINE has on fragmentation. Since EMM − ONLINE makes no use of the available information of the order of region execution, the worst case fragmentation is based on the difference in size of regions and the chosen DMA policy in the same way as for fine grain models.[29] Therefore, using $\text{RPM} - \text{Unscoped}^{\text{RPM}}_{\text{EMM-ONLINE}}$ will be of benefit over a fine grain memory model in the space dimension only if the worst case fragmentation overhead that would have been incurred in the fine grain model is more than that of fragmentation and the cost of delaying deallocation in $\text{RPM} - \text{Unscoped}^{\text{RPM}}_{\text{EMM-ONLINE}}$. The timing characteristics of EMM − ONLINE $\text{TimeReq}^{\text{RPM}}_{\text{EMM-ONLINE}}(P)$ are partially dependent on the chosen DMA and partially on the added overhead of EMM − ONLINE. Fewer calls to the DMA means that the time overheads for the whole program are reduced. However, at each inter-region location, EMM − ONLINE must be invoked in order to pass calls to the DMA or update that region's bitmap.

4.4.2 The EMM − OFFLINE Memory Subsystem

EMM − OFFLINE takes advantage of existing research into offline allocation strategies [94, 82, 67] based on the information described in the LSL. For programs in $\text{RPM}^{\rightarrow}$, the lifetime intervals required by this analysis are trivial to divulge: for a region r_i with size $\mid r_i \mid$ and deallocation point r_j^+, the lifetime interval can be described as a pair (i, j). For programs with fixed loops in $\text{RPM}^{\circlearrowleft}$,

[29] The assumption here is that the number of regions is still greater than the difference in region size[94].

loop unrolling can be employed to achieve a similar specification. However, as noted in Section 4.3.2, programs in RPM$^?$ can currently not take full advantage of this analysis. Infinite loops can be dealt with in this model by employing loop unrolling up to the point where the relative lifetimes of regions follow some fixed pattern such that every iteration of a region can be allocated in the same position.

Existing offline allocation strategies do not consider multiple threads as the relative lifetime of regions in different threads can not be determined. A trivial solution for programs in RPM$^{\|}$ and other multi-threaded models without region sharing and synchronisation is to consider each thread independently and reserve an area of memory for each thread equal to that derived in the offline analysis of that thread. With the introduction of cross-thread region sharing however, this option is no longer possible. Crucially, although making use of the information expressed through region sharing and synchronisation were shown to require lower memory requirements in the LSL, existing analysis can not carry this benefit down to the memory subsystem in a way that is implicitly done by an online memory subsystem such as EMM − ONLINE. In the case of the former, this is because the exact lifetime of a region in a thread that this analysis depends on is unknown; in the case of the latter it is because existing memory sharing analysis does not consider multiple flows. Although such offline analysis could be developed, it would be expected to be non trivial. A memory subsystem can be developed in which regions in the same thread share the same area of memory except for those regions involved in region sharing. These latter regions are placed in a separate area of memory that uses an online DMA, meaning that existing offline analysis can be used on the remaining regions.[30] Unfortunately, this solution therefore fails to take full advantage of either region sharing or region synchronisation.

4.4.3 An Offline Strategy Without Pre-defined Allocation Points

The constrained versions of the RPM provide a mechanism for investigating another research niche that is not considered in existing offline allocation strategies: the possibility of changing *when* memory for a region is allocated and deallocated rather than *where* memory for that request is allocated. The advantages and disadvantages of such an approach are considered in more detail in Section 7.2. It suffices to show here that changing when memory for a region is allocated and deallocated may prove beneficial in reducing the worst case memory requirements of a program as fragmentation is reduced or eliminated. As an example, consider the fragmentation example given in Section 2.1.1 on page 15. If after Step (8) a new object of size 3 units is created, then the memory requirements of the whole program would be 13 units as none of the unit size spaces at Step (8) would suffice for this object. By observing when objects are deallocated in this program, the EDPs of these objects would be based on the deallocation points shown in Figure 4.19 (a). Rather than allocating

[30] A similar solution is adopted for a memory subsystem of RPM − Scoped and is described in more detail in Section 5.8.

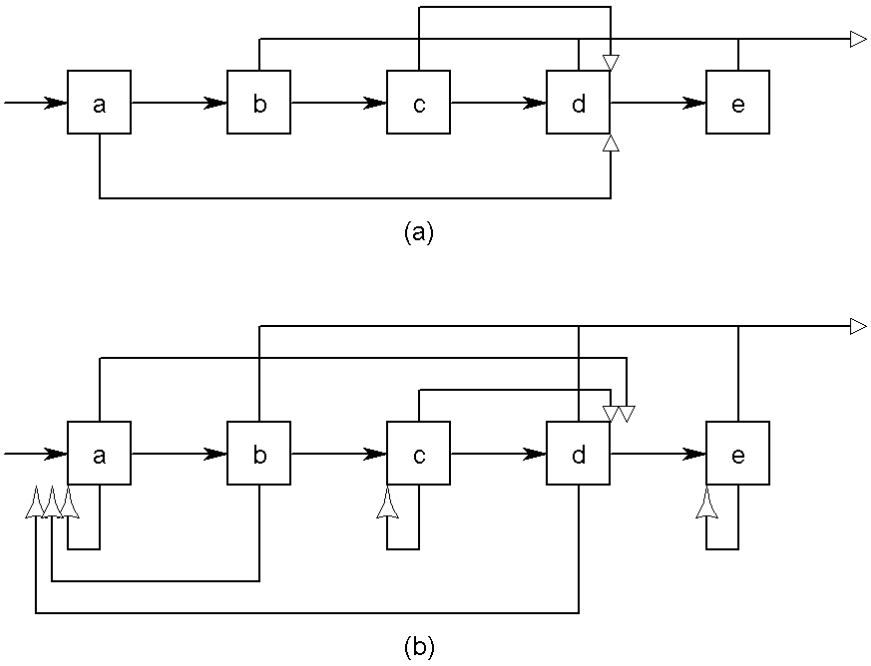


Figure 4.19: Preallocating Memory using RPM Notation

each object at its LAP and deallocating it when specified in its EDP, consider an alternative strategy where objects are allocated and deallocated in the order as shown in Figure 4.19 (b) and for which RAPs and RDPs can therefore be described. Using these new allocation and deallocation points, the order of execution and allocation and deallocation becomes that shown in Figure 4.20. Observe that in this latter figure, before the execution of (a) even begins, 9 units of memory have already been allocated. At each step, the memory for a region is guaranteed to already have been allocated. At the last step, three units of contiguous memory blocks are still available for a new object. Therefore, the worst case memory requirements for this program have been reduced from 13 units to only 11 units by arranging allocation and deallocation in an order which eliminated fragmentation. This order is in fact a scoped order derived automatically using an algorithm that will be developed in the next chapter.

4.5 Summary

In this chapter, a Lifetime Specific Definition for coarse grain memory models called the Region Partitioning Model was introduced. The model RPM$^{\rightarrow}$ was developed first as a mechanism for expressing the deallocation points of regions in programs made of sequentially executing regions in a single thread. A constrained version of RPM$^{\rightarrow}$, cRPM$^{\rightarrow}$ was developed that allowed the expression of an allocation point for a region to be at an arbitrary point in the program before that region

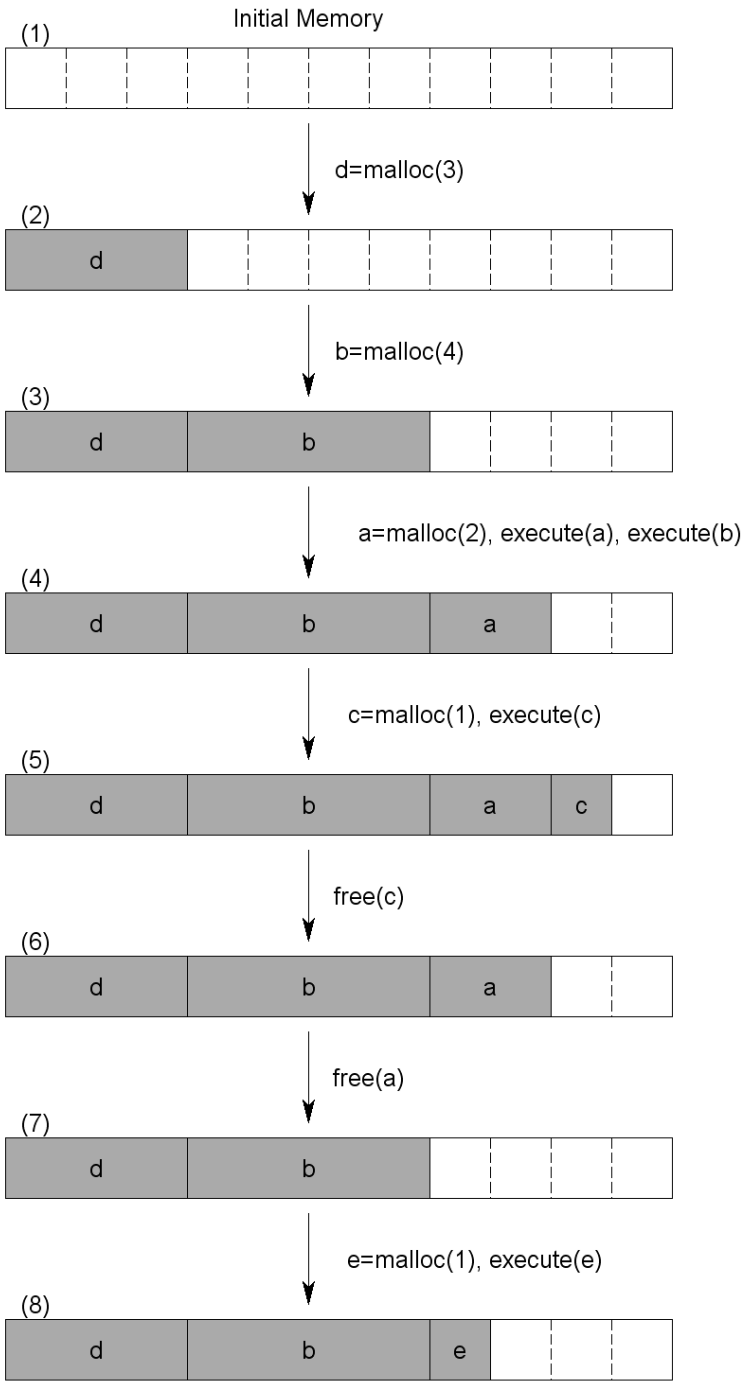


Figure 4.20: Eliminating Memory Fragmentation Using a Scoped Approach

was to be executed, rather than immediately before it. An alternative deallocation point to that described for $\text{RPM}^{\rightarrow}$ programs could also be specified as long as it occurred after this latter point. A set of criteria called the "correctness criteria" were defined in order to validate the prescribed allocation and deallocation points. Equations for calculating the memory requirements of programs expressed in these models were derived. $\text{RPM}^{\rightarrow}$ and $\text{cRPM}^{\rightarrow}$ were then extended to consider other control flows including iteration and selection. The correctness criteria and memory requirements of these models were also derived. Multi-threading was introduced for programs where each thread is made up of sequentially executing regions as in $\text{RPM}^{\rightarrow}$ and $\text{cRPM}^{\rightarrow}$. The impact of region sharing where objects in a region in a thread reference objects in a region in another thread was investigated. Also, exposing synchronisation between threads to the memory model was shown to be useful in reducing the worst case memory requirements at the level of this specification.

A simple memory model called $\text{RPM} - \text{Unscoped}$ was introduced to illustrate how a complete memory model can be implemented by plugging in any memory subsystem that can interpret the lifetime information specified in a given LSL of the RPM. It was argued that using a coarse grain model with a memory subsystem which makes no use of the information expressed on region lifetime would still be subject to fragmentation. Indeed, using this model in applications where the difference in the size of regions is large would derive no benefit from a coarse grain approach in terms of the worst case space requirements. In contrast, a memory subsystem that performs an offline allocation strategy can take advantage of the information specified in the STS. However, although online DMAs automatically leverage information on region synchronisation and region sharing to derive lower worst case memory requirements, it is not clear how similar benefits can be leveraged in DMAs that use offline algorithms to derive an allocation strategy. Finally, it was shown that reorganising allocation and deallocation in a scoped order can be used to reduce the worst case memory requirements of a program by eliminating fragmentation. The next chapter will use three models of the RPM, $\text{RPM}^{\rightarrow}$, $\text{RPM}^{?}$ and a subset of $\text{RPM}^{\circlearrowleft}$ to explore such algorithms that derive new allocation and deallocation points from user-specified ones. This allows an investigation of where such an approach is beneficial in reducing the worst case space requirements of a program.

Chapter 5

The RPM – Scoped Memory Model

The different versions of the Region Partitioning Model (RPM) introduced in Chapter 4 are Lifetime Specification Languages (LSLs) that allow the developer to describe the lifetime of regions in a Spatio-Temporal Specification (STS) and then feed this information as input to the memory subsystem that implements that model. This chapter investigates how such an STS in three versions of the RPM ($RPM^{\rightarrow}$, $RPM^{?}$ and a subset of $RPM^{\circlearrowleft}$) could be used to derive another STS that conforms to the constraints of a scoped model. The source is therefore a program description given in the unconstrained versions of the RPM models and for which the allocation point of a region immediately precedes it (that is it lies at its Latest Allocation Point (LAP)) and the deallocation points are those specified in the Earliest Deallocation Point (EDP) specification. The target is an annotated program by which the semantics of these annotations specify the Real Allocation Points (RAPs) and Real Deallocation Points (RDPs) as described in the constrained version of the same model. These latter allocation and deallocation points are derived by searching for valid annotations of the program that satisfy the constraints of the scoped model. In addition to the advantages of automating this process, using the memory requirement expressions for the constrained versions of these models makes it possible to derive the memory requirements of these programs before considering the additional mechanism-related overheads of the memory subsystem. This allows an evaluation of the scoped approach to coarse grain memory management and motivates extensions to this model that address any shortcomings.

This chapter begins with a detailed investigation into the memory model RPM – $\text{Scoped}^{\text{cRPM}^{\rightarrow}}$ that applies the constraints of the scoped model to programs in $RPM^{\rightarrow}$. Therefore, from a program specification in an unconstrained version of $RPM^{\rightarrow}$ for which EDPs are specified, a program in $cRPM^{\rightarrow}$ is derived. This is achieved by deriving regions' allocation and deallocation points such that these conform to a scoped structure and then expressing these points using RAP and RDP notation. A similar investigation is carried out for two other versions of the RPM: $RPM^{?}$ and a subset of $RPM^{\circlearrowleft}$ that contains no conditional loops or infinite&conditional loops. It is shown that programs

with infinite loops in RPM$^{\circlearrowleft}$ can have EDPs specified such that no scope ordering exists for which memory requirements are finite in cRPM$^{\circlearrowleft}$ where they were finite in RPM$^{\circlearrowleft}$. This therefore proves the second part of the thesis hypothesis.[1] The implications of region sharing and synchronisation are investigated in the context of the multi-threaded version of the RPM, RPM$^{\|\sigma\rho}$. Finally, suitable memory subsystems for RPM – Scoped that leverage the advantages of the information expressed in these models are proposed.

5.1 RPM – Scoped **with LSL** cRPM$^{\rightarrow}$**:** RPM – Scoped$^{\text{cRPM}^{\rightarrow}}$

The LSL for RPM – Scoped described in this section is based on a memory model that describes programs in cRPM$^{\rightarrow}$. Using the notation introduced in Section 3.4, this model is therefore denoted as RPM – Scoped$^{\text{cRPM}^{\rightarrow}}$. The RTSJ exports an API for using scoped memory that centres around two methods: *enter()* and *executeInArea()*. The LSL for RPM – Scoped$^{\text{cRPM}^{\rightarrow}}$ also makes use of these two methods in order to specify the RAPs and RDPs of regions. In the RTSJ, the memory for executing a region is created and, on *enter()* or *executeInArea()* being invoked on these memory regions, the backing store represented by these regions becomes the current allocation context for all objects created by the thread. In RPM – Scoped$^{\text{cRPM}^{\rightarrow}}$ however, there is no separation between the region and its backing store. Therefore, when *enter()* is invoked, the memory for a region is created at that point. The RTSJ also requires implementations to enforce a single-parent rule that dictates how regions must be ordered. The algorithms developed in this chapter make scope ordering of regions automatic and guarantee that the single-parent rule is always enforced. Therefore, an implementation would no longer require to execute this check. By using a combination of *enter()* and *executeInArea()* invocations, the algorithms derive a scope structure that minimises the memory requirements of a program whilst respecting the allocation and deallocation order dictated by the scoping rules.

The semantics of *enter()* and *executeInArea()* within the context of the cRPM$^{\rightarrow}$ are defined first. Recall that RPM$^{\rightarrow}$ and cRPM$^{\rightarrow}$ are the single threaded and sequential versions of the RPM. As the program definition of this model has no stack representation to describe entering and leaving an invocation, these methods are symbolised by four annotations: **ENTER**, **EXIT**, **EIA-ENTER** and **EIA-EXIT** to describe respectively entering and exiting a memory area, starting execution in a memory area on an *executeInArea()* method and finally exiting when *executeInArea()* returns. These annotations are used to define the lifetime semantics of regions between the annotation pairs. A rule governing valid annotations also ensures the proper nesting of annotations.

Given a sequence of regions $r_{\overrightarrow{i,j}}$ in a program in RPM$^{\rightarrow}$, an annotation pair **ENTER** and **EXIT**

[1] To reiterate, this states that it can be proven that the strictly scoped model fails under certain conditions with memory requirements being infinite.

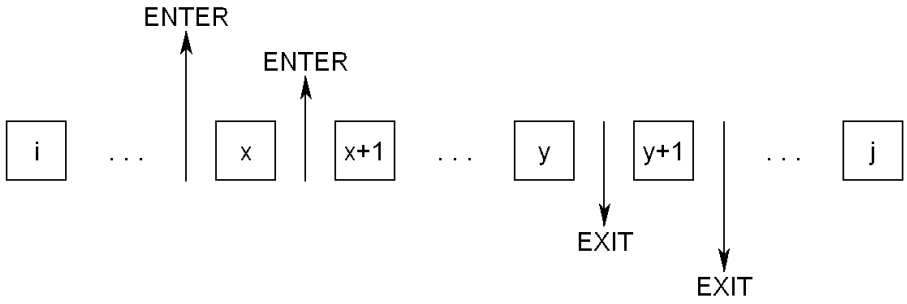


Figure 5.1: Nested **ENTER**/**EXIT** Annotation Pairs

can be placed around $r_{\overrightarrow{i,j}}$ so that if $S = r_{\overrightarrow{i,j}}$, then S can be split into three ($S = ABC$). If $A = r_{\overrightarrow{i,x}}$, $B = r_{\overrightarrow{x+1,y}}$ and $C = r_{\overrightarrow{y+1,j}}$, then if is possible to place an **ENTER** annotation at r^-_{x+1} and an **EXIT** annotation at r^+_y (and for which A and/or C can be empty). Alternatively, a new annotation can be added to a region sequence that already contains previous annotation pairs. Consider the first of these two cases. Placing this annotation pair implies that all regions in B have a RDP[2] at earliest at r^+_y and an RAP at latest at r^-_{x+1}. The use of "earliest" and "latest" is necessary here as further annotations can further constrain the lifetime of these regions as described for the second case below. Figure 5.1 shows how nested **ENTER**/**EXIT** annotation pairs are depicted throughout this thesis.

Consider the second of the two cases listed above where an annotation is added to a region sequence that already contains a previous annotation pair. In this case, all regions already enclosed in the existing annotation pairs retain their existing allocation and deallocation point bounds. The remaining regions form a set with each region having an allocation point at latest at the location of the new **ENTER** annotation and a deallocation point at earliest at the location of the new **EXIT** annotation. In order to guarantee proper nesting in a partitioning, a valid annotation addition to an STS is allowed only if the new annotation pair encloses an equal number of previously existing **ENTER** and **EXIT** annotation pairs. Indeed, if this were not the case then the lifetime of regions could be ambiguous by such a specification as it could contradict the earliest/latest semantics specified in a previous annotation. All regions are initialised with an allocation point at the start of the program and a deallocation point at the end. As more annotation pairs are added, the possible set of allocation and deallocation points becomes further constrained. Once the final annotation pair is added, the latest possible allocation point of a region becomes the single element in the RAP specification of that region and the earliest possible deallocation point becomes the single element in the RDP specification of that region. Placing an annotation pair must therefore satisfy two constraints: firstly, the addition of an new annotation pair must follow the nesting rule; secondly, the lifetimes

[2]The superscript identifying the memory model $\mathrm{RPM-Scoped}$ and the LSL $\mathrm{cRPM}^{\rightarrow}$ shall henceforth be suppressed in this section; therefore RDP implies $\mathrm{RDP}^{\mathrm{RPM-Scoped}^{\mathrm{cRPM}^{\rightarrow}}}$ and $\mathrm{MemReq}(r_x)$ implies $\mathrm{MemReq}^{\mathrm{RPM-Scoped}^{\mathrm{cRPM}^{\rightarrow}}}(r_x)$ unless specified otherwise.

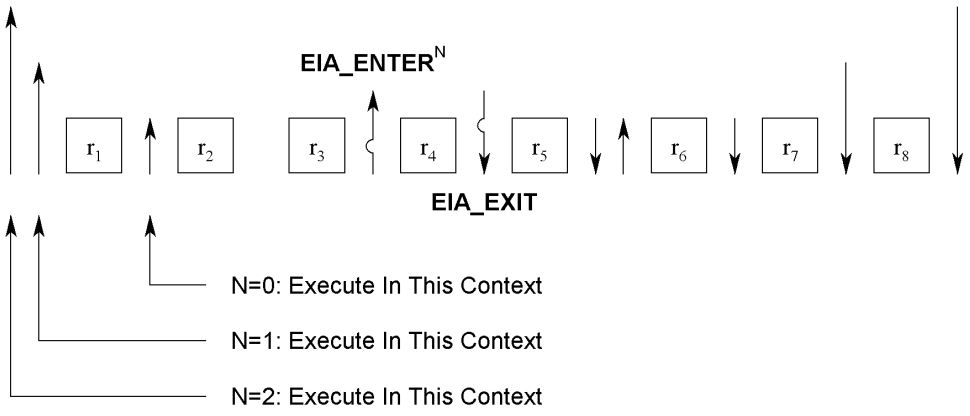


Figure 5.2: Using Indexes to Identify allocation contexts in *executeInArea()*

of the regions defined by the annotation must be correct. If the original program was correct in that all EDPs were correct, then correctness in cRPM$^{\rightarrow}$ is verified if the RAPs and RDPs of regions derived from the annotations are correct in cRPM$^{\rightarrow}$.

executeInArea() differs somewhat from that of *enter()* in that a specific memory area within the current scope stack must be identified as the location which is to become the allocation context for that region. One way to do this is to further annotate **EIA-ENTER** to **EIA-ENTER**^{N} where N is some positive integer that shows that the regions between the annotation pair are to be executed N positions up the memory stack. The example in Figure 5.2 illustrates how this index is used to identify the allocation context for executing region r_4. The straight arrows represent **ENTER** and **EXIT** annotations with arrows of the same length representing a pair. The allocation contexts to execute r_4 for different values of N are identified. The semantics of *executeInArea()* are defined as follows: Given an annotation pair **EIA-ENTER**^{N}**/EIA-EXIT** enclosing regions r_i to r_j, let X be the position of the **ENTER** annotation at index N and Y be the position of the pairing **EXIT**. For all regions r_x in $r_{\overrightarrow{i,j}}$, $\mathrm{RAP}(r_x) = X$ and $\mathrm{RDP}(r_x) = Y$. In practice any annotation of **EIA-ENTER** that identifies the allocation context for that region can be used. For example, in an implementation this could be a direct reference to the memory area allocated for that region in the corresponding **ENTER/EXIT** annotation.

The STS for RPM $-$ Scoped$^{\mathrm{cRPM}^{\rightarrow}}$ programs as described above can be generated automatically from RPM$^{\rightarrow}$ programs by providing a transformation from an STS that is correct in RPM$^{\rightarrow}$ to one which satisfies the constraints of RPM $-$ Scoped$^{\mathrm{cRPM}^{\rightarrow}}$ and is correct in cRPM$^{\rightarrow}$. This transformation is in essence a constraint-satisfaction problem and a number of possible solutions can be derived. In particular, the case in which a single **ENTER/EXIT** annotation pair is placed respectively at r_1 and r_n of a program $P = \langle r_{\overrightarrow{1,n}}, <, \mathfrak{E} \rangle$ will give a valid RPM $-$ Scoped$^{\mathrm{cRPM}^{\rightarrow}}$ program $P = \langle r_{\overrightarrow{1,n}}, <, \mathfrak{E}, \mathfrak{A}, \mathfrak{D} \rangle$ for which all regions r_i in $r_{\overrightarrow{1,n}}$ will have a specification $\mathrm{RAP}(r_i) = r_1^-$ in $\mathfrak{A}$

and $\text{RDP}(r_i) = r_n^+$ in $\mathfrak{D}$. Trivially, this annotation respects the scoping rules of annotations and is correct in cRPM$^{\rightarrow}$. However, the identified annotation should require the minimum possible memory requirements that satisfy the scoping constraints while still being correct in cRPM$^{\rightarrow}$. The next two sections develop algorithms that carry out this transformation automatically.

5.2 Algorithms for Scope Ordering using *enter()*

The annotation process for the STS of RPM – Scoped programs effectively aims to restrict the allocation and deallocation points of the regions between the **ENTER** and **EXIT** points. Assume a program $P = \langle r_{\overrightarrow{1,n}}, <, \mathfrak{E} \rangle$ in RPM$^{\rightarrow}$ and for which every region has some arbitrary deallocation point specified in the EDP of that region. The algorithm for minimising memory requirements when using *enter()* alone is given in Algorithm 2. By placing an **ENTER** annotation just before r_i, the allocation point of r_i becomes r_i^-, the latest and optimum position. The lifetime semantics of an *enter()* annotation require that the pairing **EXIT** be placed at some position r_j^+ where $r_j^+ > r_i$ such that all regions between r_i and r_j^+ have a deallocation point specified in their EDP at r_j^+ as the earliest possible position. This annotation is therefore placed at the first legal position allowed by these semantics as this is the earliest point at which the enclosed regions can be deallocated. If $\text{EDP}(r_i)$ satisfies this constraint, then the **EXIT** annotation is placed in this position. In this case, $\text{RDP}(r_i) = \text{EDP}(r_i)$ and the memory for r_i will exist only for as long as it is required. However, if some region r_j between r_i and $\text{EDP}(r_i)$ has a deallocation point specified in its EDP at some point after $\text{EDP}(r_i)$, then it is no longer possible to place the **EXIT** annotation at $\text{EDP}(r_i)$ since this would imply that this region has an RDP earlier than its EDP, therefore breaking the correctness criteria of RDPs. Instead it will have to be placed at the first point r_y^+ such that all regions between r_i and r_y^+ have an EDP at earliest at r_y^+. By Equation 4.6, all regions r_p where $\text{EDP}(r_i) < r_p < r_y^+$, incur the additional memory cost of size $\mid r_i \mid$ in $\text{MemReq}^{\text{RPM-Scoped}^{\text{cRPM}^{\rightarrow}}}(r_p)$ over that in $\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_p)$. The algorithm proceeds until an annotation pair has been made for all regions. Having identified the inter-region location at which to place an **ENTER** or **EXIT** annotation, the algorithm ensures that this annotation is placed in the correct position amongst any other existing annotations in order to satisfy the nesting rules. Since an **ENTER/EXIT** annotation pair is placed for every region in this algorithm, the location of the **ENTER** annotation becomes the allocation point specified in the RAP of that region and the **EXIT** annotation becomes the deallocation point specified in the RDP of that region. Therefore, every **ENTER/EXIT** annotation pair will only be relevant to a single region. **ENTER** annotations are therefore further annotated with the region for which they hold in order that the correct memory size for that region is allocated. **EXIT** annotations do not need such further information attached to them as they will always represent a deallocation point for the region at the top of the scope stack.

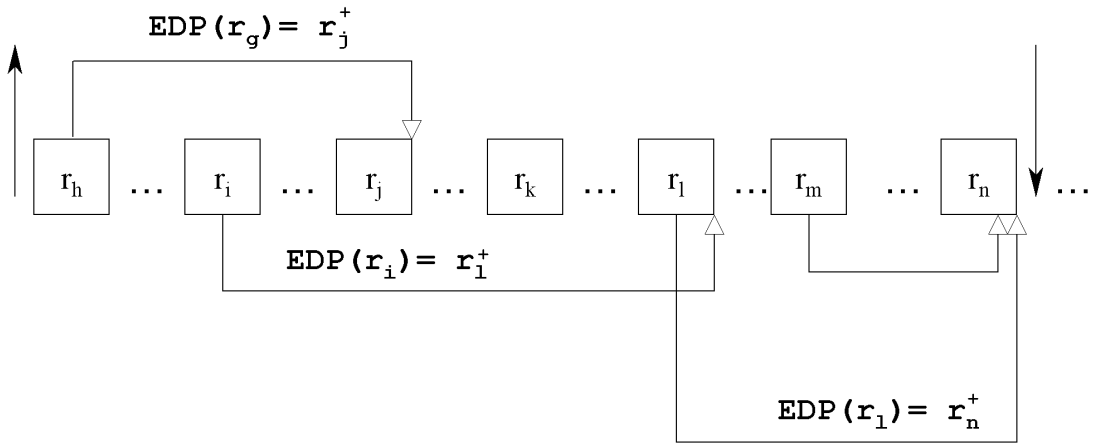


Figure 5.3: Finding an **ENTER**/**EXIT** Annotation Pair for a region

As an example, consider Figure 5.3 and the processing of region r_h. First, an **ENTER** annotation is placed at r_h^- to mark the allocation point for this region. Although the EDP for this region is at r_j^+, an **EXIT** annotation cannot be placed at this point as the EDP of r_i which lies between r_h and r_j comes after r_j^+. The next position to test for the pairing **EXIT** annotation is therefore at the deallocation point of r_i which is at r_l^+. Once again, the region r_l prohibits a legal **EXIT** annotation at this point. Finally, the EDP of r_l, r_n^+, is tested and, since all regions between r_h and r_n have EDPs that also fall in this range, the pairing **EXIT** annotation for the **ENTER** annotation at r_h^- is placed at this point.

Algorithm 2 will find the ideal stack ordering allowed using *enter()* alone. The algorithm is optimal by construction as each region is allocated at its LAP and deallocated as early as is legal under the semantics of **ENTER** and **EXIT**. This algorithm is illustrative and more efficient algorithms could be developed. For example, a more efficient algorithm can be achieved by processing regions from the end of the program and at each stage maintaining a list of valid **EXIT** points. It is interesting to note that this set will consists of all regions r_i for which $\text{EDP}(r_i) = r_i^+$. If the only element of this set is the last region in the program, then the memory required for the whole program would be equal to the sum of the memory requirements of all regions in the program.

5.3 Algorithms for Scope Ordering using *enter()* and *executeInArea()*

An Algorithm that uses both *enter()* and *executeInArea()* to find the best ordering for regions is developed next. Let the program P be a program with n regions all of which have some arbitrary

Data : P: `Program` = $r_{\overrightarrow{1,n}}$: `RegionSequence`
Result: AnnP: `AnnotatedProgram` =AnnSeq: `AnnotatedRegionSequence`
foreach $r_i \varepsilon P$ **do**
  Place **ENTER** in AnnP at r_i^-
  r_j^+: `InterRegionLocation`=first position where $r_j^+ > r_i \wedge \forall\, r_x \varepsilon r_{\overrightarrow{i,j}}, \text{EDP}(r_x) <= r_j^+$
  /*`Having found where to place the EXIT, correct nesting needs to be ensured` */
  if $\exists$ *a number of* ***EXIT*** *annotations* r_j^+ **then**
    Add **EXIT** in AnnP at r_j^+ such that number of **EXIT**s before the new **EXIT**=number of **ENTER**s after r_i
  else
    Add **EXIT** in AnnP at r_j^+
  end
end
return AnnSeq=AnnP

Algorithm 2: annotateWithEnter(P): Algorithm for annotating RPM − Scoped Programs using *enter()*

deallocation point specified in their EDP specifications. Consider region r_1 with $\text{EDP}(r_1) = r_j^+$. An **ENTER** annotation is placed at r_1^- for which a pairing **EXIT** annotation must be found. Ideally, this would be at r_j^+ so that r_1 could be allocated and deallocated at the earliest possible points. Let r_g be some region where $r_1 < r_g <= r_j$ and $\text{EDP}(r_g) = r_m^+$ and $r_m^+ > r_j^+$. Without *executeInArea()*, Algorithm 2 would place the **EXIT** annotation at r_m^+. This will delay the deallocation of r_1 and increase $\text{MemReq}^{\text{RPM-Scoped}^{\text{cRPM}^{\rightarrow}}}(r_x)$ where r_x represents all regions between r_j^+ and r_m^+ by $\mid r_1 \mid$ over that in $\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_x)$. The memory for r_g, however, can still be allocated and deallocated at the ideal positions and therefore r_g would not contribute to increasing the memory requirements of any region r_k over that in $\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_k)$. With *executeInArea()* there is now an alternative: The pairing **EXIT** for the **ENTER** at r_1^- can still be placed at r_j^+ whilst r_g is wrapped in an **EIA-ENTER**^{N}/**EIA-EXIT** annotation pair as represented by the curved arrows. The ideal position in which to execute r_g must now be identified by placing a new **ENTER** and **EXIT** annotation pair which will be indexed by **EIA-ENTER**. The ideal position is that which places the allocation point of r_g as late as possible and before r_m and the deallocation point as early as possible after $\text{EDP}(r_m)$. Given that an **ENTER**/**EXIT** pair has been placed at r_1^- and r_j^+ respectively, a new legal annotation pair can be placed at best at r_1^- and r_m^+. The **EIA-ENTER** is then set to index this location. In this way, $\text{RDP}(r_g) = r_m^+$ which is ideal in that r_g is deallocated at the point specified in its EDP but $\text{RAP}(r_g) = r_1^-$ meaning that r_g needs to be allocated earlier than it is actually used. As shown in Figure 5.4, this will increase $\text{MemReq}^{\text{RPM-Scoped}^{\text{cRPM}^{\rightarrow}}}(r_x)$ where r_x represents all regions between r_1^- and r_g^- by $\mid r_g \mid$. The memory for r_1, however, is allocated and deallocated at the ideal positions and therefore r_1 will not contribute to increasing the memory requirements of any region

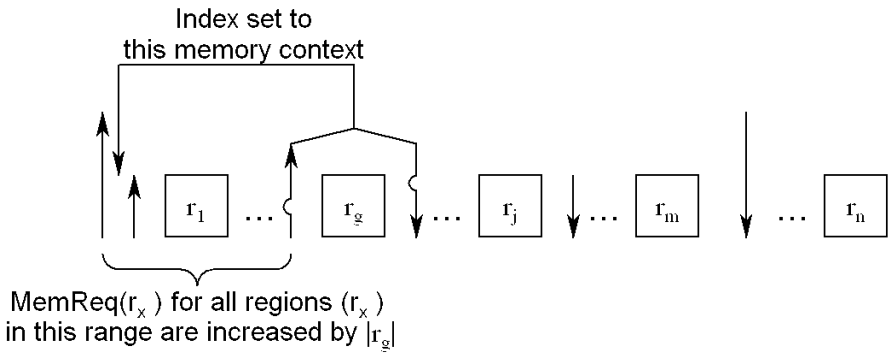


Figure 5.4: An Annotation Solution using *executeInArea()*

r_k over that in $\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_k)$. This example shows that there are two possible options, one of which must be selected. The algorithm for finding the minimal use of memory follows: that option is chosen which in increasing the respective memory requirements as described incurs the smallest increase (if any) to $\text{MemReq}(P)$.

The algorithm as described so far makes two assumptions: firstly, there is only one region between r_1 and r_j^+ which has a deallocation point specified in its EDP beyond r_j^+; secondly, all regions between r_j^+ and r_m^+ have a deallocation point by r_m^+ at the latest, implying that an **EXIT** placed at r_m^+ is legal. The algorithm needs to be generalised to remove these assumptions. For clarity, each assumption will be addressed in turn. Let the program to be annotated be $P = \langle r_{\overrightarrow{1,z}}, <, \mathfrak{C} \rangle$. Let S be an ordered set of all the regions between r_1 and $\text{EDP}(r_1) = r_j^+$ which have an EDP after r_j^+. Figure 5.5 illustrates this case with an example in which there are three regions with a deallocation point after r_j^+: r_g, r_h and r_i. These are ordered in S with the region with the first deallocation point coming first in the list. Therefore, in this case $S = \{r_i, r_g, r_h\}$. The second assumption is maintained for now and it is therefore assumed that all the regions before r_l, r_m and r_n have deallocation points at or before r_l^+, r_m^+ and r_n^+ respectively. Rather than only the two annotation options previously possible, there are now four possible annotations as illustrated in Figure 5.6. The first pairing **EXIT** for the **ENTER** placed at r_1^- can be placed either at $\text{EDP}(r_1)$ (which in this case is r_j^+) or at a deallocation point specified in the EDP of each of the regions in S.[3] In each case, the regions in S with a deallocation point before this **EXIT** annotation can be allocated after r_1 and deallocated before this **EXIT**, possibly at their ideal locations. However, all regions before this **EXIT** annotation and after r_j^+ will require $\mid r_1 \mid$ more memory. Also, for each region r_x of the regions in S that have a deallocation point specified in the EDP that comes after this **EXIT**, all

[3]These points are the only points that need to be tested for as all other regions between r_2 and r_n (when maintaining the second assumption), placing an **EXIT** annotation after any other region r_x than those specified as deallocation points in the set S would require r_x to be allocated earlier (at r_1^-) without the benefit of some other region being able to be allocated later.

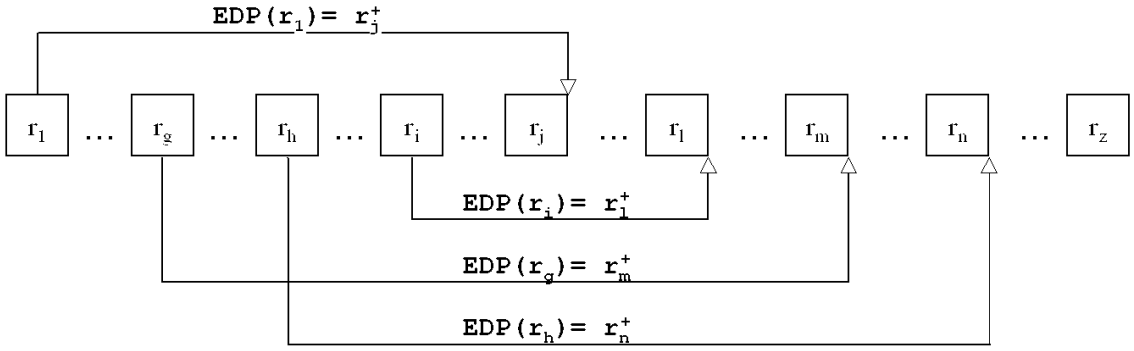


Figure 5.5: Removing Assumption (1) in Annotation Assignment: An Example

regions r_k between r_1 and r_x (r_1 inclusive and r_x not inclusive) will require $\mid r_x \mid$ more memory in $\text{MemReq}^{\text{RPM}-\text{Scoped}^{\text{cRPM}^{\rightarrow}}}(r_k)$ than in $\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\rightarrow}}}(r_k)$.

The algorithm proceeds with the remaining region sequences being processed in turn as sequences of regions with the region that is wrapped in the annotation for *executeInArea()* removed from the sequence. For example in Solution 2 in Figure 5.6 the sequences to be processed are $r_{\overrightarrow{2,l}} - \{r_g, r_h\}$, $r_{\overrightarrow{l+1,m}}$, $r_{\overrightarrow{m+1,n}}$ and $r_{\overrightarrow{n+1,z}}$. The possible annotations that result from processing of the first three sequence are then tested for their memory requirements when used in conjunction with this solution. The combination that leads to the minimum possible requirements in comparison to the other solutions used in conjunction with their resulting possible annotations is chosen as the annotation for the region sequence $r_{\overrightarrow{1,n}}$.

Consider the example in Figure 5.7 which shows a region sequence $r_{\overrightarrow{d,n}}$ in program $P = \langle r_{\overrightarrow{a,z}}, <, \mathfrak{E} \rangle$ and the EDP of some of the regions in the sequence. Using the algorithm described so far, the only element in S for region r_a would be r_g. With the first **ENTER** annotation placed at r_d^- and the pairing **EXIT** at r_j^+, r_g would be wrapped in an *executeInArea()*. This time however, a new element r_k inhibits a legal allocation context with an **ENTER** at r_d^- and **EXIT** at r_m^+. Once again there are two options available: either move this **EXIT** annotation to r_n^+ (and thereby delaying the deallocation of r_g), or make use of *executeInArea()* on region r_k (and thereby allocating r_k early). In general this situation can occur several times during the processing of a region, and the complexity of the search space needs to be analysed. This investigation is carried out in Section 5.3.2 below.

In order to identify all the possible annotations that need to be derived and checked in the final algorithm, the implications of moving an **EXIT** annotation to the right must be considered. The solutions for the example in Figure 5.5 were obtained by moving the **EXIT** annotation to the right so that at each stage the deallocation point of r_1 as would be specified in the RDP was delayed but more regions got a chance to be allocated closer to their LAP rather than at r_1^-. The final position

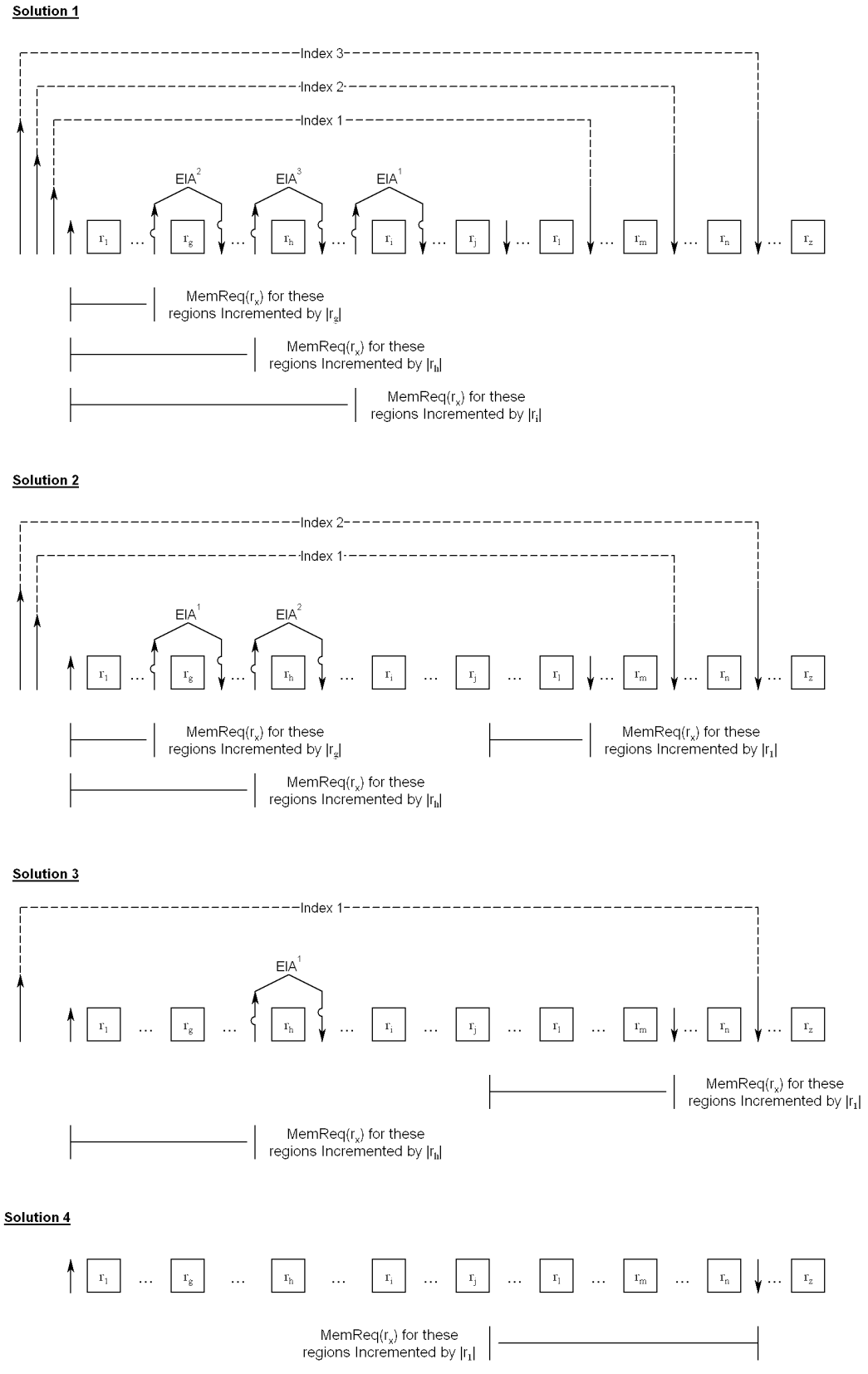


Figure 5.6: Removing Assumption (1) in Annotation Assignment: Enumerating Solutions

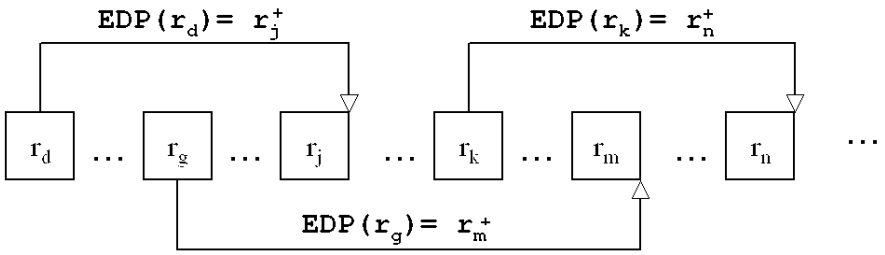


Figure 5.7: Removing Assumption (2) in Annotation Assignment

tested was the position at which further delaying the deallocation of r_1^- would have derived no benefit as all regions r_i between the annotation pair had a deallocation point specified in EDP(r_i) to be at latest at the point of the last **EXIT** annotation. The set S for a region r_i therefore identifies all the points at which delaying region deallocation of r_i region could result in the benefit of some other region being allocated at a later point. By ordering this set by the EDP of these regions immediately provides a proper nesting as required for the regions that need to be wrapped in an **EIA-ENTER/EIA-EXIT** annotation pair. The region sequence from the region r_i used to derive the set S of r_i to the EDP of the last region in the set S for r_i is called a *chain*.

The algorithm for generating S is shown in Algorithm 10 in Appendix B. For some region r_i the algorithm gives a set of regions ordered by EDP, the first of which is always r_i as EDP(r_i) is the first valid location for an **EXIT** that needs to be checked when processing r_i. The procedure *SortByEDP(S)* sorts the set S by EDP with that region with the first EDP coming first. Algorithm 11 creates the global variable *SSets* that contains the set S for each region and is used in the main algorithm for finding the best annotation. The algorithm for finding the best annotation that could give the minimum requirements for P is given in Algorithm 6 (*annotateWithEIA()*) which calls Algorithm 7 with P. Algorithm 7 finds the first chain of the input region sequence and calls Algorithm 8 to find the best annotation for that chain before calling itself recursively with the remaining region sequences. The annotations for the resulting annotated chains are string concatenated together using the function *JoinChains()*. Algorithm 8 takes a chained region sequence and generates all possible annotations for that sequence, calling Algorithm 7 where necessary to find the best annotation for subsequences of the input sequence. Once all annotations have been generated, the best annotation is chosen using the function *SelectBestAnnotation()* that calculates the memory requirements of each annotation and selects that with the smallest cost. Function *SelectBestAnnotation()* is an implementation of Equation 4.8 in Section 4.1. In order to generate the annotations, Algorithm 8 creates clones of the input sequence and calls Algorithm 9 to insert the new annotations when *executeInArea()* is to be used.

5.3.1 The Correctness of RAP and RDP Specifications Derived through *annotateWithEnter(P)* and *annotateWithEIA(P)*

The Algorithms *annotateWithEnter(P)* and *annotateWithEIA(P)* both describe a transformation from an STS for a program in RPM$^{\rightarrow}$ to an STS for a program in cRPM$^{\rightarrow}$ in which the RAPs and RDPs of regions are defined by the annotations. This transformation must be shown to derive a correct STS in cRPM$^{\rightarrow}$ if the original STS in RPM$^{\rightarrow}$ is also correct. As noted above, the allocation and deallocation point of every region in the program is initialised to be at the beginning and end of the program respectively so the allocation and deallocation points of all regions would be specified at these points in the RAP and RDP specifications. As each annotation pair is added in *annotateWithEnter(P)*, the constraints of the algorithm require all enclosed regions to have a lifetime within the set of regions enclosed in this annotation pair. Since the boundaries of these annotations set the RAP and RDP of enclosed regions, the algorithm is correct by construction since every region has an allocation point at its LAP and a deallocation point in its RDP that comes after that specified in its EDP.

A similar argument holds for *annotateWithEIA(P)*. When considering a region r_i, the derived set S of possible locations for the **EXIT** annotation of r_i is always at earliest at EDP(r_i). When the location of the **EXIT** annotation is selected, the semantics of *enter()* require that all regions in between the annotation pair have a deallocation point specified in the EDPs of these regions to be at earliest at the location of this **EXIT** annotation. For regions for which this is not the case, *executeInArea()* is used for these regions, thereby guaranteeing that the deallocation point in the RDP of each of these regions comes after that specified in the their respective EDPs. *executeInArea()* can cause a region to be allocated earlier than its LAP but not later so RAP specifications are always correct. Since at initialisation, the RAPs and RDPs of all regions are correct, and adding annotations for a complete chain is also correct then, inductively, the resulting RAPs and RDPs derived from *annotateWithEIA*(P) are also correct.

5.3.2 Understanding the Search Space of *annotateWithEnter(P)* and *annotateWithEIA(P)*

The two algorithms *annotateWithEnter(P)* and *annotateWithEIA(P)* are optimal in the sense that the best possible annotation is found that gives the minimum memory requirements that satisfy the correctness criteria of cRPM$^{\rightarrow}$ and that conform to the scoping rules. This is achieved by calculating the memory requirements of all possible combinations of annotations that could lead to these memory requirements being smallest. When using *enter()*, each region's allocation point as specified in the RAP is fixed at the LAP of that region (as identified by the **ENTER** annotation) and only a valid deallocation point location for which the RDP is specified needs to be found (as

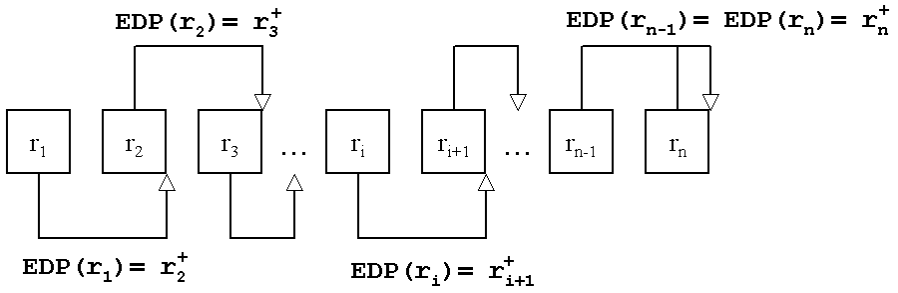


Figure 5.8: The Program with the Worst Case Search Space for an Optimal Annotation

identified by the **EXIT** annotations once the algorithm has completed execution). The search space is small as there is only one possible location for an **EXIT** annotation that is ideal - that which comes earliest and for which all regions have a deallocation point specified in their EDPs that comes before this annotation. The time complexity of *annotateWithEnter(P)* as given in Algorithm 2 would be $O(n^2)$ if a sequential search is made to identify the first valid location for **EXIT**. As described above, the algorithm can be as simple as finding those regions r_i with $\text{EDP}(r_i) = r_i^+$ and then iterating through each region in turn and selecting one of these locations to place the **EXIT** annotation for that region. In this case, the time complexity is linear.

When using *executeInArea()*, the search space is much larger when looking for the optimal annotation. Every region can have an allocation point specified in its RAP at any point before that region and choosing the best deallocation point therefore requires consideration of the effect this has on the other regions. The space and time requirements for finding the best stack ordering using *annotateWithEIA(P)* depends on the number of elements in the set S found for every region using Algorithm 10 and the number of region sequences generated every time a region sequence is analysed. In the best case, each region has only one element in this set (itself), thus reducing the problem to a solution for which *enter()* invocations suffice. In the worst case, every region r_i has every element succeeding r_i in the set. This program is depicted in Figure 5.8 and occurs when the EDP of every region r_i (except the last) occurs at r_{i+1}^+. In this case the search space grows in a non-polynomial way. Since each chain is effectively a partitioning of the ordered S-set of a region, the number of ways that this can be done is given by the Catalan Number [44] of the number of elements in this set. This number is small if the program is made up of less than about 10 regions ($\text{C}(5) = 14$,$\text{C}(10) = 4862$) but for programs with more regions, the search space becomes too large ($\text{C}(50) > 1.9 \times 10^{27}$, $\text{C}(100) > 8.9 \times 10^{56}$). Nevertheless, *annotateWithEIA(P)* lends itself to a several heuristics that can contain the search space albeit at the cost of a non-optimal solution. An example of such a heuristic is described next.

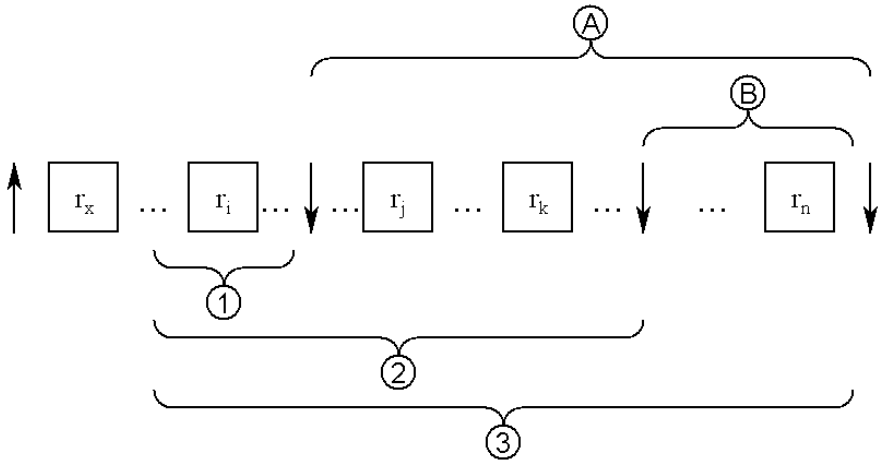


Figure 5.9: Using Heuristics to Reduce the Complexity of *annotateWithEnter(P)*

Recall that the S-set for a region r_i gives the positions at which delaying region deallocation of one region could be beneficial if some other region can be allocated at a later point. Therefore, rather than searching all of these possible positions, a subset of the elements of S could be investigated. The choice of which elements of this set are analysed can be based on an approximation of the memory requirements for the sequences derived in each partition of the chain. This approximation could be based on a simple averaging function over this sequence, a selection function based on the memory requirements of the partition holding the region with the greatest memory requirements, or any other heuristic function. Rather than the comprehensive depth-first approach given in Algorithm 6 for *executeInArea()*, this non-optimal solution takes a breadth first approach whereby a subset of the possible partitionings of the chain (possibly just one or maybe all) are chosen to mark out the chosen **EXIT** locations based on the heuristic evaluation of the memory requirements in the resulting subsequences.

As an example, consider the region chain in Figure 5.9 in which the downward arrows show the possible **EXIT** locations within the S-set of the region r_x. Regions r_i, r_j, and r_k are the regions that are made to execute in a nested region using *executeInArea()*. Three possible values for the memory requirements of this chain when putting the first **EXIT** annotations in the given locations and subsequent **EXIT** annotations at the end of the chain are:

1. $\max(\text{MemReq}(H^*(\mathbf{1} - \{r_i\}))+ \mid r_x \mid, \text{MemReq}(H^*(\mathbf{A})))+ \mid r_i \mid$

2. $\max(\text{MemReq}(H^*(\mathbf{2} - \{r_j, r_k\}))+ \mid r_x \mid, \text{MemReq}(H^*(\mathbf{B})))+ \mid r_j \mid + \mid r_k \mid$

3. $\text{MemReq}(H^*(\mathbf{3}))+ \mid r_x \mid$

The function $H^*(r_{\overrightarrow{i,j}})$ is a fitness function that returns heuristic value for the memory requirements of the sequence $r_{\overrightarrow{i,j}}$. The smallest value of the three possibilities is chosen as the annotation for the chain starting with region r_x and, given this annotation, the next iteration of the algorithm derives an annotation for the resultant region subsequences. Note that the algorithm used in this example partitions the chain into at most two parts as the region sequence after which the first **EXIT** location is placed is not partitioned further but a single **EXIT** annotation is placed at the end of the chain. Therefore, there are $\mid S \mid$ possible annotations that need to be investigated for the set S of a given region. For a better solution, the heuristic algorithm could search a further subset of the set of possible annotations up to some fixed depth after the first **EXIT** annotation has been placed.

5.4 RPM – Scoped **with LSL** cRPM$^?$**:** RPM – Scoped$^{\text{cRPM}^?}$

This section considers the transformation of programs in RPM$^?$ to programs in cRPM$^?$ through a similar annotation to that described in Sections 5.2 and 5.3. The memory model RPM – Scoped$^{\text{cRPM}^?}$ uses cRPM$^?$ as its LSL. The allocation and deallocation points as described in the RAP and RDP specifications are derived such that the memory requirements of the program are as small as possible while still respecting the constraints of the scoped model and the correctness criteria of cRPM$^?$.

5.4.1 Scope Ordering using *enter()* in RPM – Scoped$^{\text{cRPM}^?}$

Recall that the EDP of a region in RPM$^?$ for some memory model $\mathcal{M}$ is specified as $\text{EDP}^{\mathcal{M}^{\text{RPM}^?}}(r_i) = E$ where $E = \{T_1, T_2, \ldots, T_n\}$, each T_t is a tuple of the form $(\{C_{t,1}, C_{t,2}, \ldots, C_{t,m}\}, r_j^+)$ and each $C_{t,c}$ is a conditional region denoted c if the boolean result of executing the conditional region c is `true` and $\overline{c}$ if it is `false`. Recall also that the correctness criteria for RPM$^?$ required every region to have a deallocation point specified for every track in $\text{Tracks}(P, r_i)$, even if two tracks could share the same deallocation point. For example, in Figure 5.10, r_2 can be deallocated at r_3^+ if c_1 evaluates to `true`, at r_6^+ if c_1 evaluates to `false` and c_2 evaluates to `true`, and at r_7^+ if both c_1 and c_2 evaluate to `false`. Therefore, $\text{EDP}(r_2) = \{(\{c1\}, r_3^+), (\{\overline{c1}, c2\}, r_7^+), (\{\overline{c1}, \overline{c2}\}, r_8^+)\}$.[4] For the four possible tracks in the program, the deallocation points specified in the EDP of each region are as shown in this figure.

The annotation of a program P using *enter()* follows a similar process as that described in Algorithm 2 for RPM – Scoped$^{\text{cRPM}^{\rightarrow}}$. For every region r_i, an **ENTER** annotation is placed at r_i^-. This guarantees that r_i is allocated at the latest possible position as $\text{RAP}(r_i) = \text{LAP}(r_i) = r_i^-$. For every track ρ in $\text{Tracks}(P, r_i)$ and every region in ρ, an **EXIT** annotation is placed at r_j^+

[4] Once again, superscripts are suppressed for EDP, RAP and RDP specifications.

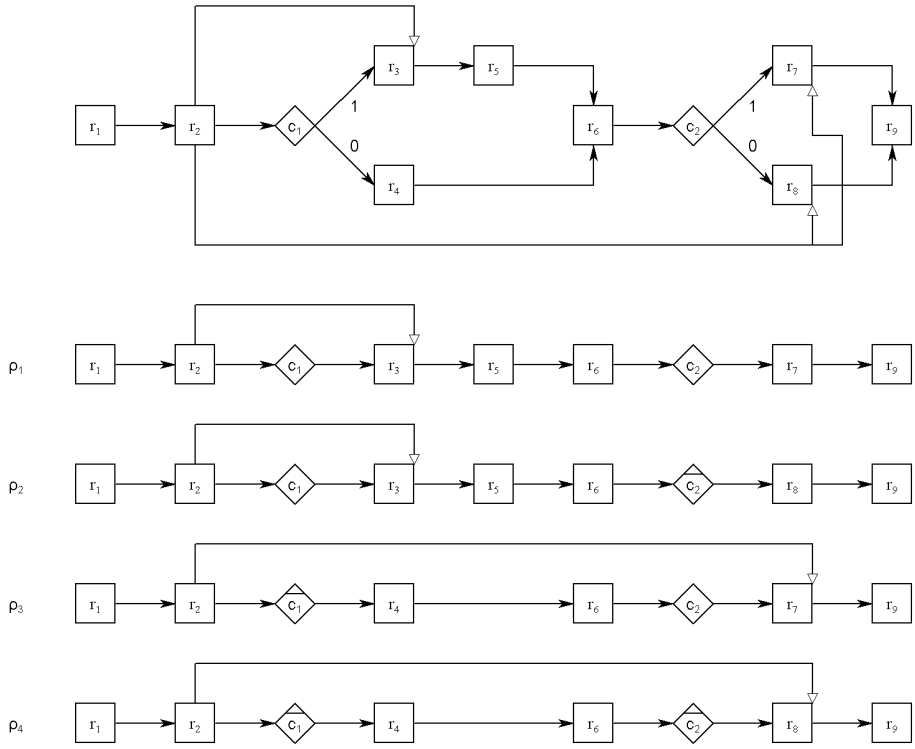


Figure 5.10: An Example of Deallocation Points on Tracks in cRPM$^?$

where $r_j^+ > r_i$ in ρ and all regions between r_i and r_j^+ in ρ have a deallocation point for that track at r_j^+ at the earliest. Therefore, although every region has exactly one **ENTER** annotation, it will have as many corresponding **EXIT** annotations as the number of tracks on which it lies. Therefore, *annotateWithEnter(P)* can be used for deriving the annotations for programs in RPM$^?$ by executing *annotateWithEnter(ρ)* for every ρ in Tracks(P). Although a single **ENTER** annotation is maintained at each r_i^- to mark the allocation point of r_i and for which no further information has to be attributed to this annotation, all derived **EXIT** annotations have to be themselves annotated with information that informs the memory subsystem of which region and track they each hold for. Note that every deallocation point must be paired with the single element in the RAP in an (α, β) pair in order to conform to the notation used for RDPs in cRPM$^?$. The derived deallocation points in cRPM$^?$ are specified using RDP notation for which the pairs in β are specified by listing the conditionals that lead to a deallocation point together with the identified inter-region location. For example, if the deallocation point of r_2 in track ρ_1 occurs at r_7^+, then one of the pairs in β in RDP$(r_2) = \{(r_2^-, \beta)\}$ is $(\{c1, c2\}, r_7^+)$.

5.4.2 The Correctness of RDP Specifications when using *annotateWithEnter(P)* on Tracks for Programs in RPM$^?$

The transformation of EDP specifications for a program in RPM$^?$ as described in Section 5.4.1 must derive RDP specifications that are correct in cRPM$^?$. Recall that if $\rho \varepsilon \text{Tracks}(P, r_i)$, then there *must* exist *exactly one* T_t in the EDP or RDP specification for r_i where T_t is in the form $(\{C_{t,1}, C_{t,2}, \ldots, C_{t,m}\}, r_j^+)$ for which all of $\{C_{t,1}, C_{t,2}, \ldots, C_{t,m}\}$ are in the regions in ρ. Therefore, if a deallocation point for a region r_i as specified in the EDP of some track is moved to some other point in this transformation so that new conditional regions exist between r_i and this point, then these conditional regions (either in the form c or $\overline{c}$) must also be found between r_i and the derived deallocation points on the other tracks that contain this regions. For example, if $\text{EDP}(r_3) = \{(\{c2\}, r_7^+), (\{\overline{c2}\}, r_8^+)\}$ in Figure 5.10, then the deallocation point of r_2 in ρ_1 as identified by the **EXIT** annotation can be at earliest at r_7^+. If the deallocation point of r_2 in ρ_2 were to remain at r_3^+ then the RDP specification would be incorrect as $\text{RDP}(r_2) = \{(\{c1, c2\}, r_7^+), (\{c1\}, r_3^+), \ldots\}$ and therefore two elements of the set $\text{RDP}(r_2)$ identify points on the same track. It can be shown however that this situation never occurs as whenever a deallocation point for a region r_i on a track has new conditional regions added between r_i and this deallocation point during the transformation, then the annotation process of other tracks that shared the first deallocation point will also add at least the same conditional regions. Continuing from the previous example, it can therefore be shown that if the deallocation point of r_2 in ρ_1 is moved to r_7^+ during the transformation, then the deallocation point of r_2 in ρ_2 must come after $\overline{c_2}$. The proof that this transformation leads to RDP specifications that are correct in cRPM$^?$ is outlined next. This proof is based on the assumption of an original specification that is correct in RPM$^?$.

Let the EDP of a region r_i be $\text{EDP}(r_i) = E$ where $E = \{T_1, T_2, \ldots, T_n\}$, and each T_t is a tuple of the form $(\{C_{t,1}, C_{t,2}, \ldots, C_{t,m}\}, r_j^+)$. Let $\Xi(T_t)$ be the subset of tracks in $\text{Tracks}(r_i, P)$ which are identified by one element T_t in E. For example, in Figure 5.10, $\Xi(\{c1\}, r_3^+) = \{\rho_1, \rho_2\}$. If the deallocation point for r_i on one of these tracks found using *annotateWithEnter()* incorporates a set X of new conditional regions in addition to those specified in T_t, then this must be because there is some region r_x between r_i and the deallocation point of r_i on that track which has a deallocation point at $r_x'^+$ and for which all regions between r_i and r_x' inclusive have a deallocation point in the EDP specification that falls before $r_x'^+$. For instance in Figure 5.10, following from the previous example, when the deallocation point of r_2 is moved to r_7^+, a new set of conditional regions ($\{c_2\}$ in this case) is included in the deallocation points for that track as specified in the RDP of r_2. This is because r_3 which lies between r_2 and r_3^+ has a deallocation point at r_7^+ and all other regions between r_2 and r_7 (both inclusive) have deallocation points before r_7^+. If the EDP specification for r_x is correct, then there can not be a deallocation point specified in the EDP for r_x which contains

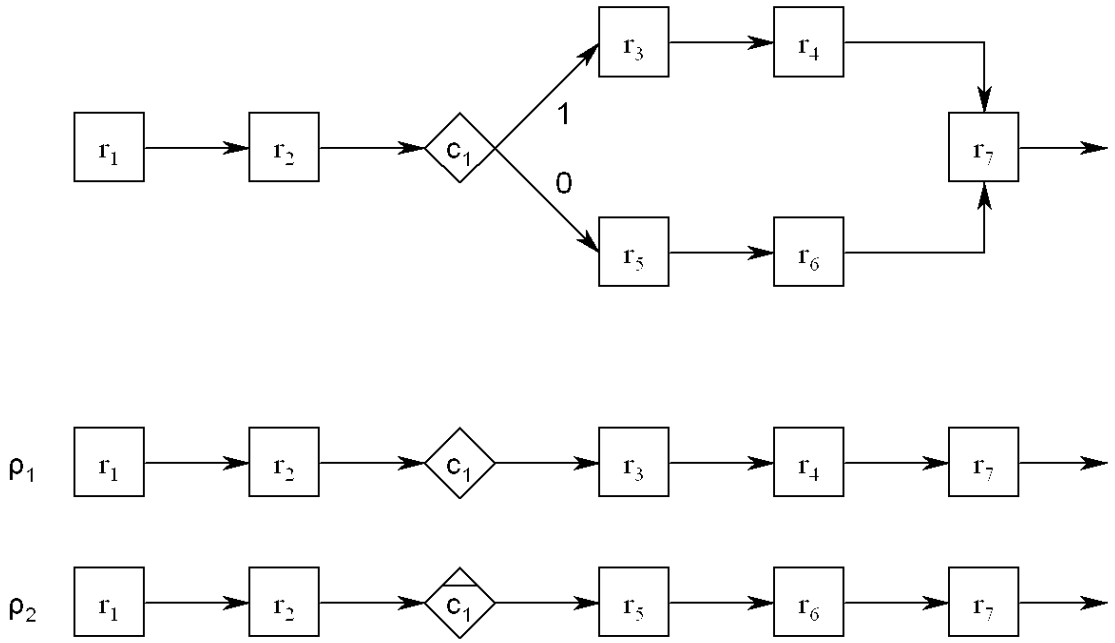


Figure 5.11: An Example of a Program in RPM$^?$ with Corresponding Tracks

only a subset of X. Therefore, for all other tracks in $\Xi(T_t)$, r_i must have a deallocation point that it moved to a location after the conditionals in X. This implies that every track in $\Xi(T_t)$ still has exactly one inter-region location that can be identified as the deallocation point for r_i on that track. For example, in Figure 5.10, if EDP$(r_3) = \{(\{c_2\}, r_7^+), (\{\}, r_6^+)\}$ then this specification is incorrect since $\{\}$ is a subset of $\{c_2\}$. The second pair of EDP(r_3) must therefore contain at least $\overline{c_2}$ in its set of conditional regions and the deallocation point for r_2 in ρ_2 must come at some point after $\overline{c_2}$ and RDP$(r_2) = \{(\{c2\}, r_7^+), (\{\overline{c2}, \ldots\}, \cdot), \ldots\}$. Hence, both ρ_1 and ρ_2, the elements of $\Xi(\{c1\}, r_3^+)$ have exactly one inter-region location that can be identified as the deallocation point on that track. Since the deallocation point specified by T_t for all tracks $\Xi(T_t)$ is expanded to a set of deallocation points for which every track in $\Xi(T_t)$ still has exactly one deallocation point, then the transformation using *annotateWithEnter()* derives RDPs that are correct in cRPM$^?$.

5.4.3 Scope Ordering using *executeInArea()* in RPM − Scoped$^{\text{cRPM}^?}$

An annotation process that uses both *enter()* and *executeInArea()* and that derives the scoped ordering with the lowest possible memory requirements is non-trivial. Whereas when using *enter()* alone it is possible to treat each track independently, this is not the case when *executeInArea()* is used. Consider the program in Figure 5.11 which contains two tracks, ρ_1 and ρ_2. If these two tracks are processed independently using *annotateWithEIA()*, then two problems could arise due to the non-oracular nature of the runtime in determining the result of conditionals.

The first problem occurs when the memory for a region is allocated but then not executed. Recall that in cRPM$^?$, each allocation point specified in the RAP of a region must be paired with a corresponding set of deallocation points that may include tracks in which that region is not executed. In RPM $-$ Scoped$^{\text{cRPM}^?}$, this means that a matching **EXIT** annotation must be made in a legal position on every track for every **ENTER** annotation. For example, in Figure 5.11, if ρ_1 is processed using *annotateWithEIA()* and r_3 is set to be allocated at r_2^- then the new **ENTER** annotation placed at this position will have no corresponding **EXIT** annotation in ρ_2. A solution to this problem would be to place a matching **EXIT** annotation in other tracks to pair this **ENTER** annotation. This **EXIT** annotation would be placed at the first possible location that observes a correct nesting and comes after a point where that region can no longer be executed. The allocation point for that region as marked by the **ENTER** annotation would then be paired in the (α, β) pair with the locations of these **EXIT** annotations in the RDP specification. However, there is no guarantee that such a location exists that obeys the scoping rules. Moreover, even if such a valid location were available, the new program annotation would impact the memory requirements of the program as calculated using Equation 4.23. Therefore, the optimal solution for a track when using *annotateWithEIA()* on that track could prove suboptimal when considering the whole program. For example, in Figure 5.11, although it may be possible to place a matching **EXIT** annotation in ρ_2 at the first possible location that observes a correct nesting and comes after $\overline{c_1}$ once the annotations for ρ_2 are found, this would impact the memory requirements of regions in ρ_2 so that the optimal solution for annotating ρ_1 could prove suboptimal when considering the whole program.

The second problem that arises if tracks are processed independently using *annotateWithEIA()* is that the same region may be set to be allocated at two or more different points that are both reachable from each other. This occurs because the allocation point for r_i might be placed at a different position by *annotateWithEIA()* for every track. For example, in Figure 5.11, r_2 might be set to be allocated at r_1^- in ρ_1 if the deallocation point of r_2 in $\rho 1$ is after the deallocation point of r_1 in this track. However, if the deallocation point of r_2 in track ρ_2 comes after that of r_1, then r_2 could be allocated at r_2^- on this track. In general, an annotation of every track in Tracks(r_i, P) using *annotateWithEIA()* could find between one and $\mid$ Tracks(r_i, P) $\mid$ different allocation points that are all reachable from each other. This means that the derived RAP specification would be incorrect as there is more than one allocation point on every track in Tracks(r_i, P). Once again, a solution to this problem might be to choose one of these allocation points (such as the earliest one) and retain the **ENTER** annotation for this region only at this point together with the paired **EXIT** annotation. However, eliminating an allocation point might leave other tracks without any allocation point at all. A similar problem that could occur is that two allocation points could be at the same inter-region location but at different nesting levels. In order to demonstrate this problem,

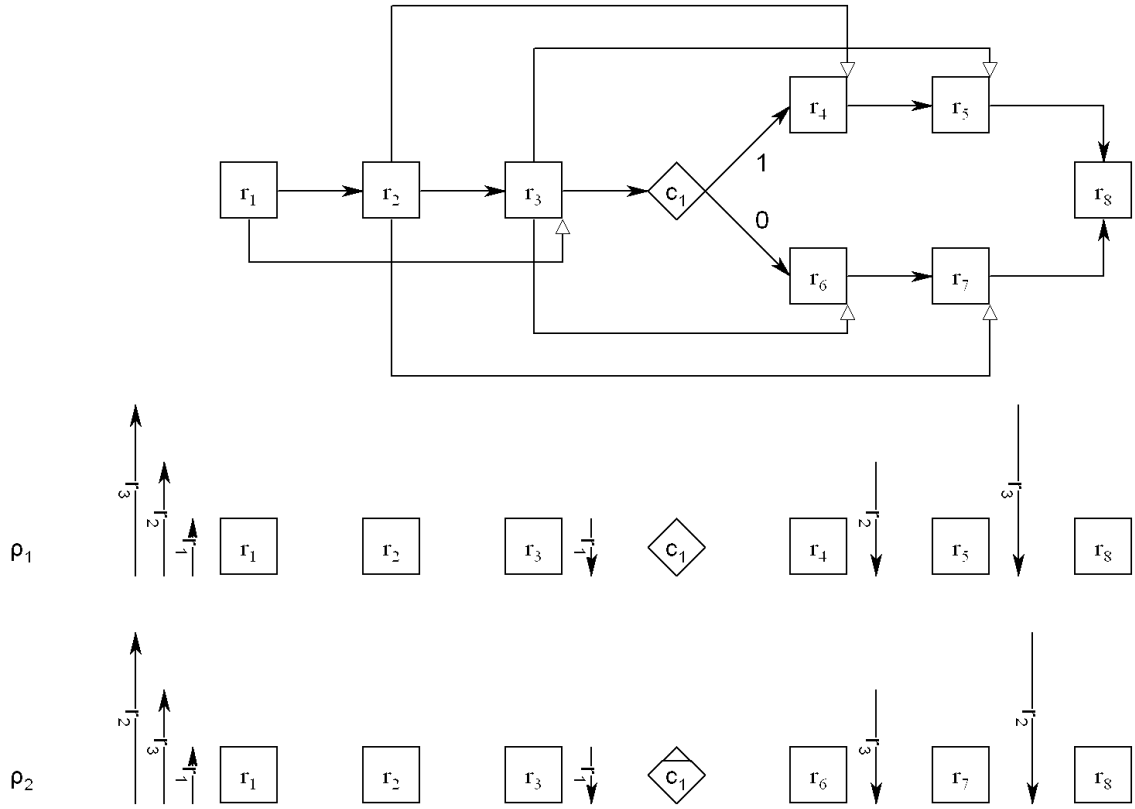


Figure 5.12: Using *annotateWithEIA()* on Programs in RPM$^?$

consider the program in Figure 5.12 with associated deallocation points and a possible annotation for the two tracks in this program. The annotation of ρ_1 places r_3 at a deeper nesting that r_2 but ρ_1 does the opposite. Even if a decision were to be taken for the location of the **ENTER** annotations (for example deciding to put the **ENTER** for r_3 before the **ENTER** for r_2 as in ρ_1), a legal position for one of the pairing **EXIT** annotations would not exist (as in this example, the pairing **EXIT** for r_2 would be at latest at r_6^+ and this is before the deallocation point of r_2 on this track as specified by EDP(r_2)). Although it is possible to also move the **EXIT** annotation for r_3 in this example in order that that for r_2 can be placed in a legal position, this would again impact the memory requirements of the whole program.

Given these two observations, it is clear that an algorithm that derives an optimal annotation for program in RPM$^?$ that respects the scoping rules, guarantees regions are not deallocated early and also gives an annotation for which no other annotation would give lower memory requirements in cRPM$^?$ would have to search a number of possibilities that would prove intractable. It would be attractive to have such an algorithm available despite the intractable search space of an optimal solution so that heuristics similar to those described for using *annotateWithEIA()* on RPM$^{\rightarrow}$ could

be developed. Such an algorithm has not be derived as part of this research and is therefore left for future work. Instead, two non-optimal alternatives are outlined next which give a valid scope ordering but without any guarantee that the final annotation gives the scope ordering with the lowest possible memory requirements. The goal here is primarily to shown how deriving such algorithms must proceed with caution in ensuring the correctness of region lifetime specifications remains correct. In particular, an evaluation of these and other non-optimal algorithms is not investigated here.

Per-Track Annotation with Annotation Shifts

A non-optimal solution can be developed by observing the two problems discussed above. This solution is based on moving annotations so that all **ENTER** annotations that fall on a track have pairing **EXIT** annotations that fall at legal positions on that track. The annotations derived using *annotateWithEIA()* on each track provide a starting point for the algorithm and annotations are shifted so that the lifetime and scope constraints are adhered to. The correctness of the final annotation is correct by construction as every annotation is checked to ensure it satisfies these constraints before the algorithm terminates. The algorithm proceeds as follows:

1. *annotateWithEIA()* is executed on every track in the program in order to provide a starting point for the algorithm. A non-annotated list of these same tracks called Θ is also available.

2. For every region r_i, a set of allocation points for r_i exist, one on each track. However, since allocation points specified in RAPs in cRPM$^?$ do not specify the path to that allocation point, it could be that several allocation points lie on the same path to r_i, albeit on different tracks. Therefore a subset of these allocation points is required such that if r_a^- and r_b^- are the set of allocation points on all tracks in the program, then r_a is not an element of the regions in Tracks(P, r_b). Any subset that satisfies this requirement is chosen arbitrarily. Note that every track in Tracks(P, r_i) still has exactly one allocation point in this subset. This allocation point is marked on every track in Θ with an **ENTER** annotation. The order of this annotation amongst other **ENTER** annotations is immaterial at this stage and will be addressed in the next step. Let r_j^- be the allocation point on one of these tracks. If r_i is a region of Tracks(P, r_j), then a pairing **EXIT** annotation is marked on Θ at the same place that *annotateWithEIA()* placed the **EXIT** annotation on that track to pair the chosen **ENTER** annotation. If this is not the case, then a pairing **EXIT** annotation is placed at the first location r_k^+ such that $\neg(r_k \prec r_i)$. r_k will always be a conditional region in this case. The pairing of each **ENTER** annotation with the set of **EXIT** annotations is recalled in order to set the (α, β) pair of the RDP specification.

3. At this point, every region in Θ has exactly one allocation point that can be reached throughout the execution of the program and these are marked with an **ENTER** annotation. Each

ENTER annotation is paired with a set of deallocation points, one for each track which is marked with an **EXIT** annotation. Although these allocation and deallocation points are correct in terms of cRPM$^?$, the nesting of these annotations might not observe the scoping rules. The next step is therefore to shift these annotations in order that the scoping rules are observed. Let r_x be any region in the program. On one of the tracks in Θ, r_x could have **EXIT** annotations between its **ENTER/EXIT** annotation pair for which the pairing **ENTER** is before the **ENTER** annotation for r_x. In this case the **ENTER** annotation for r_x is shifted to the left until all **ENTER** annotations after the **ENTER** annotation for r_x have pairing **EXIT** annotations that come before the **EXIT** annotation for r_x. Similarly, r_x could have **ENTER** annotations between its **ENTER/EXIT** annotation pair for which the pairing **EXIT** is after the **EXIT** annotation for r_x. In this case the **EXIT** annotation for r_x is shifted to the right until all **EXIT** annotations before the **EXIT** annotation for r_x have pairing **ENTER** annotations that come after the **ENTER** annotation for r_x. This process is repeated for all regions until the nesting of all annotations is correct. Note that the final annotation and therefore the memory requirements after this step will depend on the order that regions are processed at this step.

4. In carrying out Step (3), the new allocation point for r_x when shifting the **ENTER** annotation might have marked two possible allocation points for r_x that can be reached. In this case, Steps (2) and (3) are repeated.

5. Once all regions have been processed, every region in the program will have exactly one allocation point and deallocation point in every track for which the nesting order is correct. Finally, the **EIA-ENTER** and **EIA-EXIT** annotations are placed around regions for which memory is not allocated at their LAP and the indexes set accordingly.

The algorithm outlined here makes an attempt to achieve an annotation with suitably low memory requirements only in starting from an annotation that is optimal for every track. The remainder of the algorithm shifts allocation points to earlier positions and deallocation points to later positions until an annotation is found for which the correctness criteria and scoped constraints are satisfied. There are several improvements that can be achieved by analysing the effects of these shifts in order to make decisions at each step. In particular, the set of chosen allocation points in Step (2) need not be arbitrary but could consider which set is more suitable for deriving an annotation with lower memory requirements. Also, the annotation shifts in Step (3) could consider which annotations are best moved. For example, it could be shown that the annotations of that region with the lowest memory requirements would be the best candidate for shifting since the earlier allocation and later deallocation would incur the smallest additional cost to the memory requirements of the program.

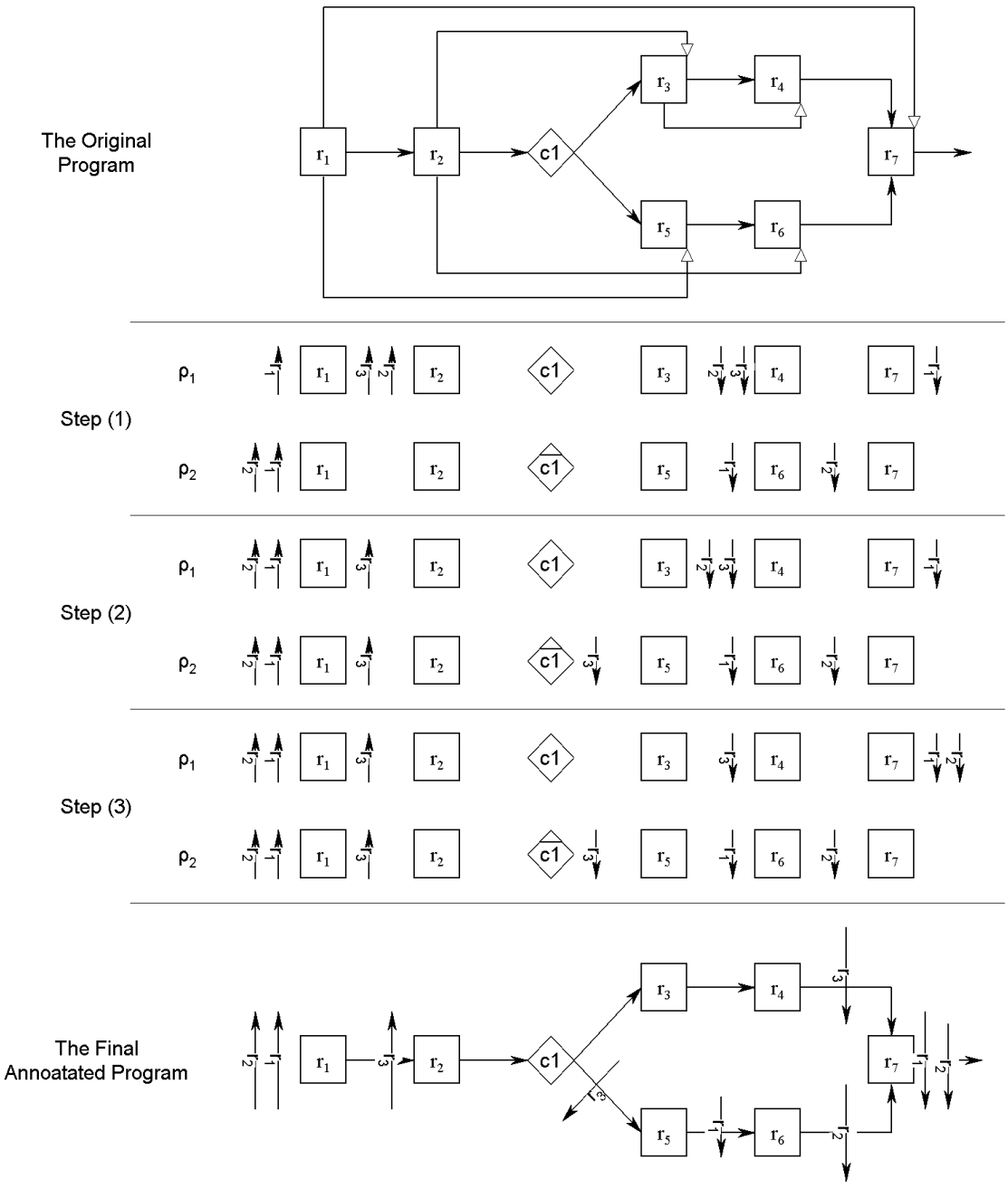


Figure 5.13: Annotating a Program in RPM$^?$ by the Per-Track Annotation with Annotation Shifts Method

As an example of how this Algorithm works, consider Figure 5.13 which shows a possible annotation for the two tracks in the program in Figure 5.11. The annotations of each track by *annotateWithEIA()* are shown and are only one possibility of all the annotations that could be used as a starting point for the algorithm. The EDPs of the regions r_1, r_2 and r_3 are as follows: $\text{EDP}(r_1) = \{(\{c1\}, r_7^+), (\{\overline{c1}\}, r_5^+)\}$, $\text{EDP}(r_2) = \{(\{c1\}, r_3^+), (\{\overline{c1}\}, r_6^+)\}$ and $\text{EDP}(r_3) = \{(\{\}, r_4^+)\}$. Assume that all other regions r_i other that r_1, r_2 and r_3 have deallocation points at r_i^+ in their EDP specification meaning that they are allocated right before and deallocated right after they are executed in every track. The arrows for the **ENTER** and **EXIT** annotations are themselves annotated to show which region the memory is allocated for at that point. Therefore, **EIA-ENTER** and **EIA-EXIT** annotations are not shown in this figure but are assumed to index the relevant positions. The per-track annotation is shown in Step (1) in this figure. In Step (2), every region has been assigned a single allocation point for every path through the program that passes through these three regions and the pairing deallocation points for each allocation point is assigned on every track. Since r_2 has an annotation pair in ρ_1 that encloses the **ENTER** annotations for r_1 and r_3 but not their corresponding **ENTER** annotations, the **EXIT** annotation for r_2 is moved to the right in Step (3) until it encloses an equal number of **ENTER** and **EXIT** annotations. At this point every region has one allocation point for every path leading to that region, a corresponding set of deallocation points for every track in the program on which an allocation point lies, and the nesting of annotations is correct. The final annotation of the program in cRPM$^?$ can be made and the RAPs and RDPs derived.

Region and Conditional Branch Coalescing

An alternative non-optimal technique for finding an annotation for programs in RPM$^?$ involves the coalescing of regions and conditional branches combined with relevant shifting of EDPs in order to derive a program in RPM$^\rightarrow$ on which *annotateWithEIA()* or a heuristic alternative can then be used. A conditional branch can be coalesced in two steps. Assuming the two branches of a conditional region do not contain any further conditional branches, the regions in each branch are merged into two regions, one for each branch. The region for each branch will have a deallocation point on a track equal to the latest deallocation point of all regions on that track and memory requirements equal to the sum of the memory requirements of these regions. For example, in Figure 5.14 (a), r_3 and r_4 with EDPs $\text{EDP}(r_3) = \{(\{c2\}, r_7^+), (\{\overline{c2}\}, r_{10}^+)\}$ and $\text{EDP}(r_4) = \{(\{c2\}, r_8^+), (\{\overline{c2}\}, r_9^+)\}$ are merged to give one new region $r_{3,4}$ with EDP $\text{EDP}(r_{3,4}) = \{(\{c2\}, r_8^+), (\{\overline{c2}\}, r_{10}^+)\}$ and $\mid r_{3,4} \mid = \mid r_3 \mid + \mid r_3 \mid$. Once there is only one region on each branch, the branch is merged into one single region. The EDP is derived in a similar way but the memory requirements are equal to the maximum of the memory requirements of the two regions on each branch. For example in Figure 5.14 (b), r_5 with EDP $\text{EDP}(r_5) = \{(\{c2\}, r_{11}^+), (\{\overline{c2}\}, r_9^+)\}$ is merged with $r_{3,4}$ to give a new region $r_{3,4,5}$ with EDP

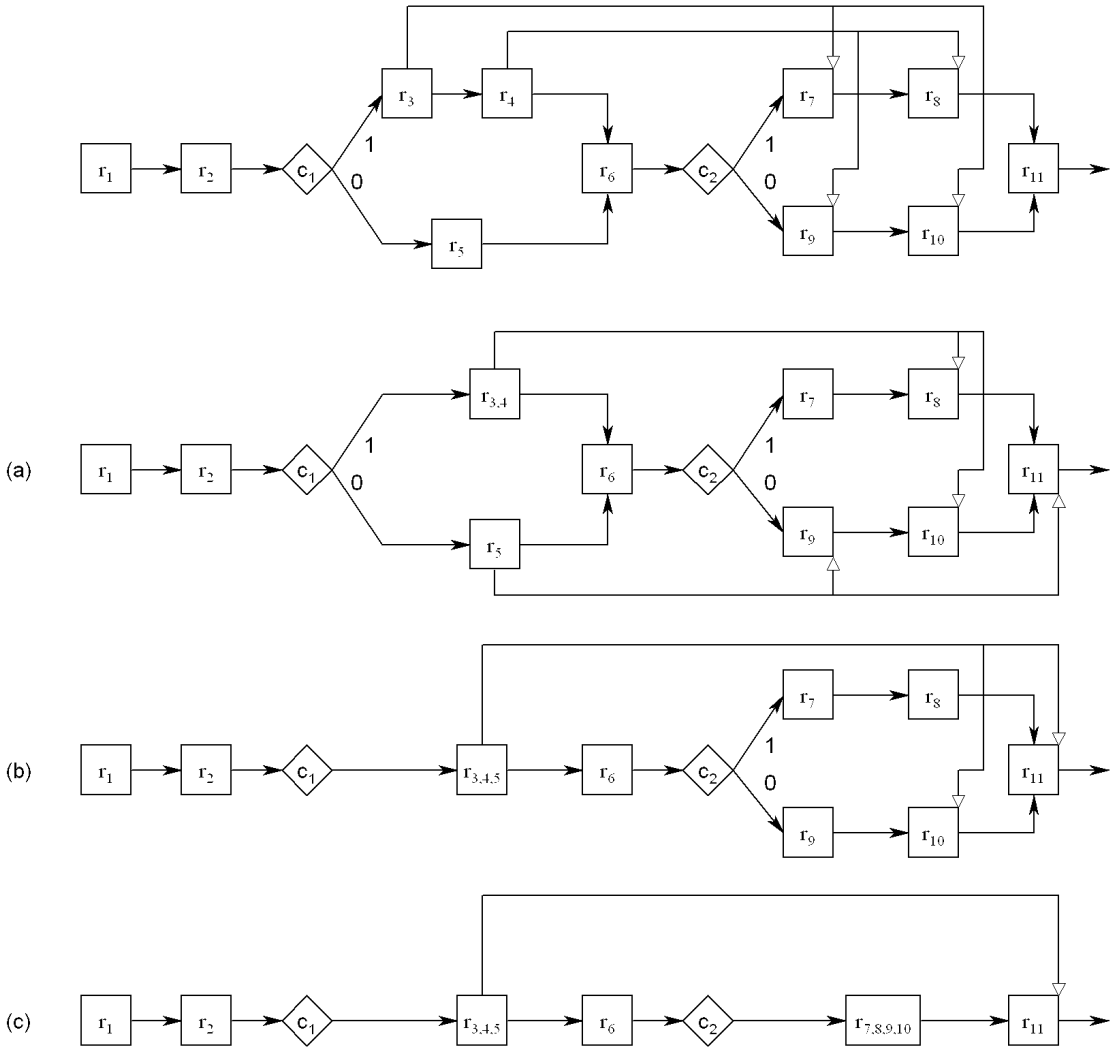


Figure 5.14: Deriving EDP Specifications in RPM$^{\rightarrow}$ from EDP Specifications in RPM$^{?}$ by the Region and Conditional Branch Coalescing Method

EDP($r_{3,4,5}$) = $\{(\{c2\}, r_{11}^{+}), (\{\overline{c2}\}, r_{10}^{+})\}$ and $\mid r_{3,4,5} \mid = \max(\mid r_{3,4} \mid, \mid r_5 \mid)$. Similarly, r_7, r_8, r_9 and r_{10} are collapsed to give the single region $r_{7,8,9,10}$. Collapsing must be done from right to left of the CFG.

Once an annotation for a program is derived using *annotateWithEIA()*, the program must be re-expanded with the derived annotations used to place suitable annotations on the original program. If some region r_j has an **ENTER** annotation right before some combined region r_i, then an **ENTER** annotation is placed at the two inter-region locations following the first two branches of the expanded annotation. For example in Figure 5.14, if r_6 is allocated at $r_{3,4,5}^{-}$ and therefore has an associated **ENTER** annotation at this point, then an **ENTER** annotation is placed at r_3^{-} and r_5^{-} in the original program. Similarly, if r_j is itself a combined region, then these **ENTER** allocations hold

for all the regions in r_j. For example, if $r_{7,8,9,10}$ has an associated **ENTER** annotation at $r^-_{3,4,5}$, then an **ENTER** annotation is also placed at r^-_3 and r^-_5 in the original program which would allocate memory of size max(| r_7 | + | r_8 |,| r_9 | + | r_{10} |). Using this approach, every region has either one or two allocation points in the whole program: if an **ENTER** annotation is placed before a regular region then it will have one allocation point; if it is placed before a combined region then it will have two allocation points. In this way, r_j has exactly one allocation point for every track in Tracks(r_j) which occurs before r_j is executed and the derived RAPs are therefore correct. For example, if $r_{7,8,9,10}$ has an associated **ENTER** annotation at $r^-_{3,4,5}$, which is then expanded to have an **ENTER** annotation at r^-_3 and r^-_5 in the original program, then RAP(r_i) = $\{r^-_3, r^-_5\}$ where $r_i \varepsilon \{r_7, r_8, r_9, r_{10}\}$.

The pairing **EXIT** annotations for the **ENTER** annotation found using *annotateWithEIA()* is similarly used to derive the **EXIT** annotations in the original program. If some region r_i has an **EXIT** annotation that comes right after some combined region r_j, then an **EXIT** annotation is placed at the two inter-region locations at which the conditional branches merge. For example in Figure 5.14, if r_7 is deallocated at $r^+_{7,8,9,10}$ and therefore has an associated **EXIT** annotation at this point, then an **EXIT** annotation is placed at r^+_8 and r^+_{10} in the original program. Similarly, if r_i is itself a combined region, then these **EXIT** allocations hold for all the regions in r_i. Again, this approach implies that every region has either one or two deallocation points in the whole program. If an **EXIT** annotation is placed after a regular region then it will have one deallocation point. If it is placed after a combined region then it will have two deallocation points. The RDP in cRPM$^?$ derived from this annotation holds for all possible paths between the region under investigation and until the location of these **EXIT** annotations. In this way, r_i has exactly one deallocation point for every track in Tracks(r_j). For example, if $r_{3,4,5}$ has an associated **EXIT** annotation at $r^+_{7,8,9,10}$, which is then expanded to have an **EXIT** annotation at r^+_8 and r^+_{10} in the original program, then the two deallocation points for r_i are $(\{c_2\}, r^+_8)$ and $(\{\overline{c_2}\}, r^+_{10})$ where $r_i \varepsilon \{r_3, r_4, r_5\}$. Since allocation points must be paired with deallocation points in cRPM$^?$, every allocation point derived as shown above is paired with each of these two deallocation points. Therefore, if RAP($r_{3,4,5}$) = $r^-_{3,4,5}$ and RDP($r_{3,4,5}$) = $r^+_{7,8,9,10}$ when using *annotateWithEIA()*, then for every r_i where $r_i \varepsilon \{r_3, r_4, r_5\}$, RDP($r_i$) = $\{(r^-_3, \{(\{c_2\}, r^+_8), (\{\overline{c_2}\}, r^+_{10})\}), (r^-_5, \{(\{c_2\}, r^+_8), (\{\overline{c_2}\}, r^+_{10})\})\}$. The shifting of EDPs to the latest possible position when coalescing regions and branches guarantees the correctness of RDPs derived this approach.

The main disadvantage of this approach is that the memory requirements of the program may be increased since the memory for a region may be allocated earlier and deallocated later than is optimal. However, this may be acceptable for programs where regions in close proximity have similar deallocation points across all paths and the number of regions in the branches of conditionals

is relatively small. Another disadvantage of this approach is that the runtime must be aware of coalesced regions and branches in order to execute each region in the correct allocation context. In addition to the reduced complexity in finding a suitable annotation for a program, an advantage of this approach is the in-built sharing of memory that comes from collapsing two branches of a conditional. A combination of this approach, memory sharing using the technique described in Section 4.3.2 and a heuristic function to calculate the memory requirements of some chains could be used. For example, region coalescing could be used to reduce the number of tracks in the program. The RAPs and RDPs of regions are then found using a heuristic version of *annotateWithEIA()* and memory sharing optimisation can then be carried out on this annotation to derive the final set of RAPs and RDPs.

5.5 RPM − Scoped **with LSL** cRPM$^{\circlearrowleft}$ **and no Conditional Loops**

In order to prove the second part of the thesis hypothesis and investigate why a scoped ordering of regions may lead to programs within infinite memory requirements, it is necessary to consider infinite programs for which a finite amount of memory is required when using a memory model such as RPM − Unscoped but for which memory requirements become infinite when applying the constraints of a scoped model. Consider a subset of RPM$^{\circlearrowleft}$ which contains only fixed loops as described in the first part of Section 4.3.3. The annotation of programs in this model can be achieved by unrolling all loops to derive a program in RPM$^{\rightarrow}$ and then executing *annotateWithEnter()* or *annotateWithEIA()*. In this way, every iteration of a region has a unique set of corresponding **ENTER** and **EXIT** annotations and therefore unique allocation and deallocation points in the RAP and RDP specifications.

A similar approach is unsuitable for programs with infinite loops as an infinite number of allocation and deallocation points would have to be specified through either an infinite set of **ENTER/EXIT** annotations pairs or a single pair in an infinite loop with a corresponding set of infinite annotations that identify every region iteration. Recall that in order to provide a finite specification, an inequality expression was introduced when describing the deallocation points in EDPs. A similar approach was also used to specify allocation points in RAPs and deallocation points in RDPs. The parallel with **ENTER/EXIT** annotations would be that an annotation pair is required which marks the allocation and deallocation point for an infinite set of regions. Using this approach, a pattern of allocation and deallocation points for a region is specified for every **ENTER/EXIT** annotation so that every one of these pairs can identify the allocation and deallocation point for an infinite number of regions. This is similar to the approach taken for eRPM$^{✠}$ described in Section 4.3.5.

Recall that in RPM$^{\circlearrowleft}$ with infinite loops, only a single infinite loop could exist, this loop had to be the outermost loop, and no regions could exist after this loop in an adjoining blocklist. Furthermore,

the deallocation point for a region in an infinite loop had to fall in that loop, albeit at an arbitrary iteration of that loop. Programs in this subset of RPM$^{\circlearrowleft}$ can be converted to a program with a single infinite loop and no other loops by unrolling all loops except for the single infinite loop. Assume for the time being that no regions prior to the infinite loop have deallocation points that fall inside the infinite loop. Using either of *annotateWithEnter()* or *annotateWithEIA()*, an annotation for all regions prior to the first region in the infinite loop can be found since the final chain of this subsequence would have to terminate at the last region prior to this loop. In considering how programs with infinite loops are annotated, it is therefore necessary to only consider programs made up of a sequence of regions wrapped in a single infinite loop.

Assume first that every region in the infinite loop I has a single inter-region location specified for all iterations of the loop. Therefore, if $\text{EDP}(r_i) = E$, then E is a set with a single element in the form $\text{EDP}(r_i, I > 0) = (r_j^+, \mathit{Expr})$. If a single annotation pair is to hold for all iterations of r_1, then these annotations would have to be themselves annotated to identify which iteration of the loop they are valid for. One way of deriving this annotation is to consider a finite number of iterations of the infinite loop and derive an annotation for this finite sequence. If it can be proven that this annotation follows a pattern that holds for arbitrary iterations of the infinite loop, then this can be represented using annotated **ENTER/EXIT** annotation pairs that describe this pattern.

As the simplest example of such a program that satisfies the previous assumption, consider a program with a single region r_1 in an infinite loop. Let $\text{EDP}(r_1, \{I > 0\}) = (r_1^+, \{I = I\})$ be the single element of E so that the I^{th} iteration of r_1 has a deallocation point right after it has finished executing at that iteration. Consider a single iteration of the infinite loop I in the previous example. Using *annotateWithEnter()* or *annotateWithEIA()*, a single **ENTER** annotation can be placed at r_1^- with a pairing **EXIT** at r_1^+. It can be shown by a trivial proof by induction that this annotation is optimal for any number of iterations of I, such that the **ENTER** annotation at the I^{th} iteration of the loop I marks the allocation point of the I^{th} iteration of r_1 and the **EXIT** annotation at the I^{th} iteration of the loop I marks the deallocation point of the I^{th} iteration of r_1. Therefore, a single **ENTER/EXIT** annotation pair can be provided in the infinite loop for which the **ENTER** annotation is itself further annotated to allocate memory for r_1 when **ENTER** is reached.[5] In this example, $\text{RAP}(r_1) = E$ and the single element in E is $\text{RAP}(r_1, \{I > 0\}) = (r_1^-, \{I = I\})$ which is paired with the single element $\text{RDP}(r_1, \{I > 0\}) = (r_1^+, \{I = I\})$.

Consider another EDP specification for this same program but with $\text{EDP}(r_1, \{I > 0\}) = (r_1^+, \{I = I + 1\})$. In order to derive suitable annotation for this program, consider that the loop I is finite and executes for n times. Therefore, $\text{EDP}(r_1, \{I = \{1..n\}\}) = (r_1^+, \{I = I + 1\})$ is specified using the shorthand notation for fixed loops. If it can be proven that the optimal annotation of this

[5]Note that the **EXIT** annotation does not have to be further annotated as it only marks the deallocation point for whichever region is at the top of the scope stack.

program is such that the memory requirements are monotonically non-decreasing with the size of the program n, then a program for which n is infinite would require an infinite amount of memory. This proof is carried out in the next section.

5.6 Proof that Memory Requirements are Infinite for some Programs in $\mathrm{RPM} - \mathrm{Scoped}^{\mathrm{cRPM}^{\circlearrowleft}}$

Let P be a program in $\mathrm{RPM}^{\rightarrow}$ where $P = \langle r_{\overrightarrow{1,n}}, <, \mathfrak{E} \rangle$ and $\mathrm{EDP}(r_i) = r_{i+1}^{+}$ $(1 \leq i < n)$, $\mathrm{EDP}(r_n) = r_n^{+}$ and all regions have equal memory requirements. This is the program that was depicted in Figure 5.8 on page 145. It will be proven by induction that any annotation for a program of size $n = 2^k$ will result in memory requirements at least of size $(k+1)\times \mid r_i \mid$. This value is therefore the lower-bound on the memory requirements of programs of this length which are annotated such that the scoping rules are observed. An example of an optimal annotation that is observed when executing *annotateWithEIA()* on this program follows:

1. An **ENTER** annotation is placed at r_1^{-} with a pairing **EXIT** annotation at $r_{2^{k-1}}^{+}$ to mark the allocation context for r_1.

2. Since $r_{2^{k-1}}$ is deallocated after $r_{2^{k-1}}^{+}$, it is wrapped in an **EIA-ENTER/EIA-EXIT** annotation. An **ENTER** annotation is placed at r_1^{-} with a pairing **EXIT** annotation at $r_{2^k}^{+}$ to mark the allocation context for $r_{2^{k-1}}$.

3. The two resulting subsequences, $r_{\overrightarrow{2,2^{k-1}}}$[6] and $r_{\overrightarrow{2^{k-1}+1,n}}$ are then processed in turn in a similar fashion. The annotation for the last two regions r_{n-1} and r_n is to place an **ENTER** annotation right before each region and the pairing **EXIT** annotation at r_n^{+}.

For example, let $k = 3$ so that $P = \langle r_{\overrightarrow{1,8}}, <, \mathfrak{E} \rangle$, $\mathrm{EDP}(r_i) = r_{i+1}^{+}$ $(1 \leq i < 8)$ and $\mathrm{EDP}(r_8) = r_8^{+}$. One possible annotation that gives the lowest possible memory requirements is as follows:

1. An **ENTER** annotation is placed at r_1^{-} with a pairing **EXIT** annotation at r_4^{+} to mark the allocation context for r_1.

2. Since r_4 is deallocated after r_4^{+}, it is wrapped in an **EIA-ENTER/EIA-EXIT** annotation. An **ENTER** annotation is placed at r_1^{-} with a pairing **EXIT** annotation at r_8^{+} to mark the allocation context for r_4.

3. The two resulting subsequences, $r_{\overrightarrow{2,3}}$ and $r_{\overrightarrow{5,8}}$ are then processed in turn in a similar fashion to give the final annotation as shown in Figure 5.15 together with a depiction of the region stack

[6] Note that *annotateWithEIA()* does not reprocess the region $r_{2^{k-1}}$ as its allocation context has already been set at this point.

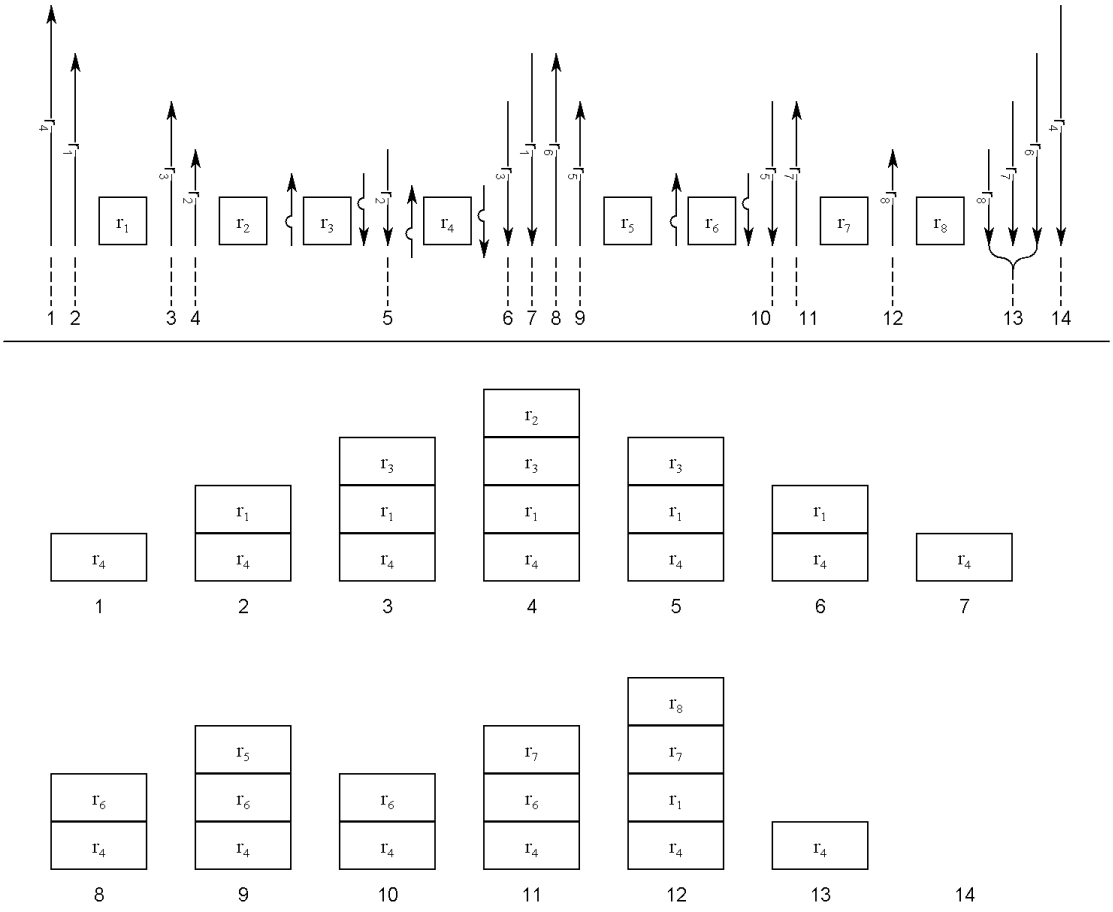


Figure 5.15: Optimal Annotation for the Program in Figure 5.8 of length 2^3

during the program execution. The memory requirements of this program are therefore equal to $4\times \mid r_i \mid$.

Note that other optimal annotations that differ to that described here may exist but the memory requirements of these annotations will always be equal to $(k+1)\times \mid r_i \mid$ for programs of size $n = 2^k$. The proof that this lower bound exists is by strong induction on the length of a program with 2^k regions. A number of lemmas are first required which are necessary for this proof.

Lemma 1: Let $P = \langle r_{\overrightarrow{1,n}}, <, \mathfrak{E} \rangle$ with EDPs and memory requirements as described above. Given that the best case memory requirements of a program of length $n = k$ are given by some value ϕ, no program of size $n = k'$ where $k' \geq k$ can have memory requirements less than ϕ.

Proof of Lemma 1: This is trivial and can be shown by contradiction. Assume that a program of size $n = k'$ where $k' \geq k$ has memory requirements bounded from below by ϕ' where $\phi' < \phi$. Since

the smaller program of size k can have as many non-functional[7] added to it as is necessary to give a program of size k', then the memory requirements of a program of size k can also be bounded from below by ϕ'. This is a contradiction since the best case memory requirements of a program of length k are given by ϕ.

Lemma 2: If ϕ memory is sufficient for a program of size $n = k$, then ϕ memory is also sufficient for a program of size $n = k'$ where $k' \leq k$.
Proof of Lemma 2: Again, this is trivial since $k - k'$ non-functional regions can be added to the program of length k'.

Lemma 3: If ϕ is the lower bound on the memory requirements for a program P of size $n = k$, then if a program P' has memory requirements less than ϕ, then P' can have at most $k - 1$ regions.
Proof of Lemma 3: This is a corollary from Lemma 1.

Lemma 4: For all $k > 0$ and L where $2 \leq L \leq k + 1$, it is always the case that $2^{k-L+1} + 2^{L-1} + 2^k - 2 < 2^{k+1}$. Equivalently, for all $k > 0$ and L where $2 \leq L \leq k + 1$, it is always the case that $2^{L-1} < 2 + 2^{k+1} - 2^k - 2^{k-L+1}$, or $0 < 2 + 2^k - 2^{L-1} - 2^{k-L+1}$.
Proof of Lemma 4: The proof is by induction on k. Let T(k) denote the inequality $0 < 2 + 2^k - 2^{L-1} - 2^{k-L+1}$ where $2 \leq L \leq k + 1$. It will be shown that for all $k > 0$, T(k) is true.
Base Case: It is first shown that T($k = 1$) is true. Therefore, since $k = 1$, then $2 \leq L \leq 1 + 1$ or $L = 2$.

$$\begin{aligned} & 0 < 2 + 2^k - 2^{L-1} - 2^{k-L+1} \\ \Rightarrow \quad & 0 < 2 + 2^1 - 2^{2-1} - 2^{1-2+1} \\ \Rightarrow \quad & 0 < 2 + 2 - 2 - 1 \\ \Rightarrow \quad & 0 < 1 \end{aligned} \tag{5.1}$$

Inductive Hypothesis: Assume that T($k = t$) is true for arbitrary $t > 0$. Therefore, it is assumed that $0 < 2 + 2^t - 2^{L-1} - 2^{t-L+1}$ where $2 \leq L \leq t + 1$.
Proof for k = t+1: It must be shown that T($k = t + 1$) also holds, that is $2^{L-1} < 2 + 2^{(t+1)} - 2^{(t+1)-L+1}$ is true for all L where $2 \leq L \leq t + 2$.

$$0 < 2 + 2^{t+1} - 2^{L-1} - 2^{t-L+2} \tag{5.2}$$

[7] A "non-functional" region is defined as one that simply allocates memory but without changing the original logic of the program.

$$\begin{aligned}
&\Rightarrow \quad 0 < 2 + 2(2^t - 2^{t-L+1}) - 2^{L-1} \\
&\Rightarrow \quad 0 < 2 + 2(2 + 2^t - 2^{L-1} - 2^{t-L+1}) - 2^{L-1} - 4 + 2.2^{L-1} \\
&\Rightarrow \quad 0 < -2 + 2^{L-1} + 2(2 + 2^t - 2^{L-1} - 2^{t-L+1})
\end{aligned}$$

Since by the inductive hypothesis $0 < 2+2^t-2^{L-1}-2^{t-L+1}$ for $2 \leq L \leq t+1$, then if $0 \leq -2+2^{L-1}$ also holds, then $\mathrm{T}(t+1)$ holds but only for values of L where $2 \leq L \leq t+1$. $0 \leq -2+2^{L-1}$ does hold since by definition $L \geq 2$ and $0 \leq -2+2^{L-1} \Rightarrow 2 \leq 2^{L-1} \Rightarrow 1 \leq L-1 \Rightarrow 2 \leq L$. It therefore only remains to be checked whether $\mathrm{T}(t+1)$ also holds for the case when $L = t+2$:

$$\begin{aligned}
& 0 < 2 + 2^{t+1} - 2^{L-1} - 2^{t-L+2} \qquad (5.3) \\
\Rightarrow \quad & 0 < 2 + 2^{t+1} - 2^{(t+2)-1} - 2^{t-(t+2)+2} \\
\Rightarrow \quad & 0 < 2 + 2^{t+1} - 2^{t+1} - 2^0 \\
\Rightarrow \quad & 0 < 1
\end{aligned}$$

Since $\mathrm{T}(k = t+1)$ holds if $\mathrm{T}(k = t)$ holds and $\mathrm{T}(k = 1)$ holds, then for all $k > 0$, $2^{k-L+1} + 2^{L-1} + 2^k - 2 < 2^{k+1}$.

Theorem: The best annotation derived using *annotateWithEIA()* on a program $P = \langle r_{\overrightarrow{1,2^k}}, <, \mathfrak{E} \rangle$ with EDPs as described above will result in memory requirements equal to $(k+1)\times \mid r_i \mid$.

Base Case: Let $P = \langle r_{\overrightarrow{1,2}}, <, \mathfrak{E} \rangle$ and $\mathrm{EDP}(r_1) = r_2^+$ and $\mathrm{EDP}(r_2) = r_2^+$. There is only one possible annotation in this case that *annotateWithEIA()* can give and it is that for which r_1 is allocated at r_1^- and deallocated at r_2^+ and r_2 is allocated at r_2^- and deallocated at r_2^+. The memory requirements in this case are equal to $2\times \mid r_i \mid$ and the base case therefore holds.

Strong Inductive Hypothesis: Assume that the best annotation derived using *annotateWithEIA()* on a program $P = \langle r_{\overrightarrow{1,2^k}}, <, \mathfrak{E} \rangle$ with EDPs as described above will result in memory requirements of size $(k+1)\times \mid r_i \mid$. Also assume that this holds for all values of k from 1 to k (the strong hypothesis).

Proof for $k+1$: It must be shown that the best possible annotation of a program $P' = \langle r_{\overrightarrow{1,2^{k+1}}}, <, \mathfrak{E} \rangle$ will result in memory requirements of size $((k+1)+1)\times \mid r_i \mid$. Equivalently, no annotation can be found using *annotateWithEIA()* with memory requirements less than $((k+1)+1)\times \mid r_i \mid$ for a program of length 2^{k+1}. Three cases are considered in turn: that for which r_1 in P' is divided into more than $k+1$ subsequences, that in which P' is not divided at all (that is one subsequence is derived as the **EXIT** annotation for r_1 is placed at $r_{2^{k+1}}^+$), and that in which P' is divided into $k+1$ or less subsequences. The proof follows by considering what the effect of partitioning this program when processing r_1 in *annotateWithEIA()* has on the memory requirements of the program in these three cases.

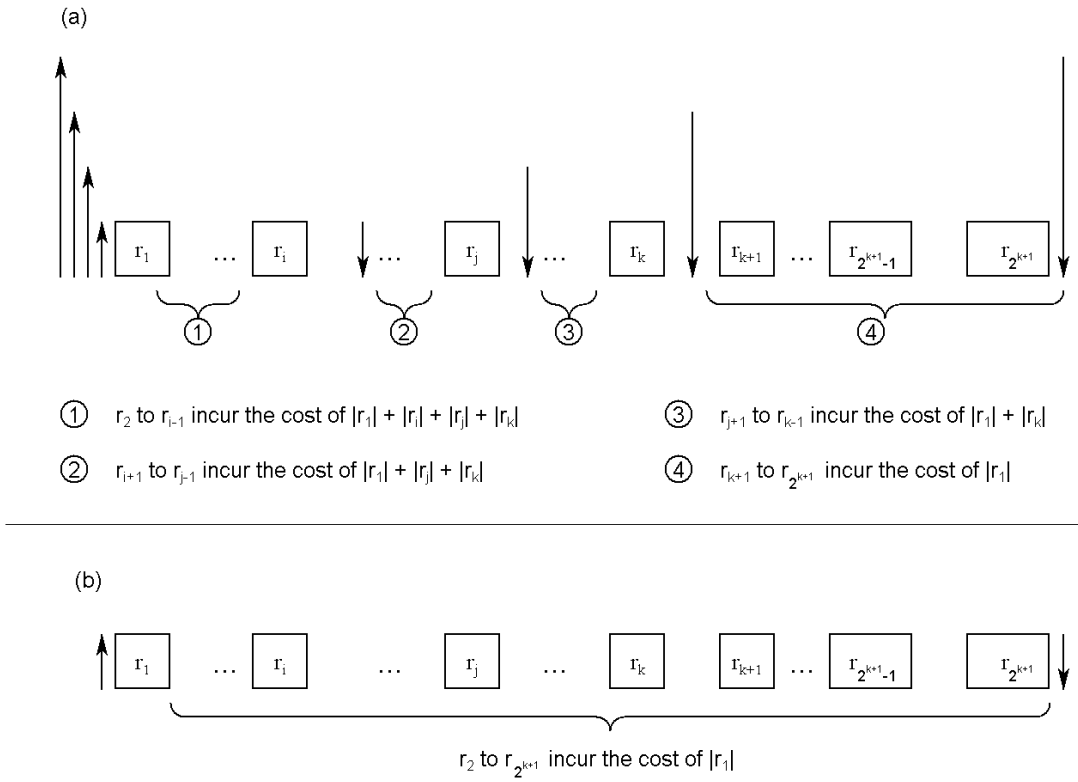


Figure 5.16: Partitioning of the Program in Figure 5.8 with Length 2^{k+1}

Case 1: More than k+1 Partitions:- As shown in Figure 5.16 (a) which depicts the first case, every time an **EXIT** annotation other than the last **EXIT** at $r^+_{2^{k+1}}$ is placed at some position r^+_i, the memory requirements of the previous subsequence are incremented by $\mid r_i \mid$. Therefore, if P' is divided into l subsequences when processing r_1, then $(l-1)$ **EXIT** annotations (excluding that at $r^+_{2^{k+1}}$) are placed in P' and the memory requirements of r_1 are increased by $(l-1)\times \mid r_i \mid$, making them equal to $l\times \mid r_i \mid$. Since $l > k+1$ for this first case, the memory requirements at r_1 are equal to at least $((k+1)+1)\times \mid r_i \mid$.

Case 2: Exactly 1 Partition:- As shown in Figure 5.16 (a), if the pairing **EXIT** annotation for the **ENTER** annotation at r^-_1 for r_1 is placed at $r^+_{2^{k+1}}$, then the remaining region sequence that needs to be processed is $r_{\overrightarrow{2,2^{k+1}}}$ which will incur the memory requirements of r_1. Since this region sequence is of length $2^{k+1}-1$ and $2^{k+1}-1 \geq 2^k$,[8] then by Lemma 1 the memory requirements of this subsequence can not be less than those of a program of size 2^k. By the inductive hypothesis the memory requirements of a program of size 2^k are at least $(k+1)\times \mid r_i \mid$. Since this sequence also incurs the memory requirements of r_1, the memory requirement of this sequence are at least $((k+1)+1)\times \mid r_i \mid$.

[8]This can be shown to be a tautology for $k \geq 0$ since $2^{k+1}-1 \geq 2^k \Rightarrow 2^{k+1}-2^k \geq 1 \Rightarrow 2^k(2-1) \geq 1 \Rightarrow 2^k \geq 1$.

Case 3: Between 2 and $k+1$ Partitions:- This third case is similar to the first case above and depicted in Figure 5.16 (a) with the difference that the number of partitions is less than $k+2$. The resulting subsequences that must be processed after r_1 are also identified in Figure 5.16 (a) by the encircled numbers. The number of subsequences is given as L (where $2 \leq L \leq k+1$) and these are ordered from the left to the right. If the **EXIT** annotation for r_1 is placed at some location r_x^+ (where $r_x < r_{2^{k+1}}$), then the sequence $r_{\overrightarrow{2,x-1}}$ is the sequence $l = 1$. This sequence incurs the cost of $\mid r_1 \mid$ and $(L-1)\times \mid r_x \mid$ or, equivalently, $L\times \mid r_x \mid$. The last partition ($l = L$) is defined as that between the $L^{th}-1$ **EXIT** annotation and the final **EXIT** annotation at $r_{2^{k+1}}$ when processing r_1. If this $L^{th}-1$ **EXIT** annotation is at r_y^+, then the sequence $r_{\overrightarrow{y+1,2^{k+1}}}$ incurs the cost of $\mid r_y \mid$. Between the first subsequence ($l = 1$) and last subsequences ($l = L$) are $L-2$ subsequences. As shown in Figure 5.16 (a), subsequence l where $(2 \leq l \leq L-1)$ incurs the additional cost of $(L-l+1)\times \mid r_i \mid$.

Assume that the memory requirements for this case are less than $((k+1)+1)\times \mid r_i \mid$. Therefore, although the memory requirements for a program of size 2^k are bounded from below by $(k+1)\times \mid r_i \mid$ by the inductive hypothesis, a program of size 2^{k+1} is not similarly bounded from below by $((k+1)+1)\times \mid r_i \mid$. If this is the case, then all the L subsequences described above have memory requirements that are less than $((k+1)+1)\times \mid r_i \mid$. By Lemma 2 and since all regions have the same size, a memory of size $(k+1)\times \mid r_i \mid$ suffices for a program of size 2^{k+1}.

Since the memory requirements derived when processing the sequence $l = 1$ incur the additional cost of $L\times \mid r_x \mid$, then this sequence must have memory requirements of less that $((k+1)-L)\times \mid r_i \mid$. Since by the inductive hypothesis a program of length 2^{k-L} has memory requirements of at least $2^{k-L+1}\times \mid r_i \mid$, then by Lemma 3, the maximum length of the sequence $l = 1$ is $2^{k-L+1}-1$. Similarly, since the memory requirements derived when processing the sequence $l = L$ incur the additional cost of $\mid r_i \mid$, then the maximum length of the sequence $l = L$ is $2^k - 1$. Also, since the memory requirements derived when processing every sequence l where $2 \leq l \leq L-1$ incur the additional cost of $(L-l+1)\times \mid r_i \mid$, then the maximum length of the sequence l is $2^{L-l}-1$. Adding region r_1 and the $L-1$ regions before each **EXIT** annotation except the last, the largest possible number of regions in P' is equal to:

$$
\begin{aligned}
& l + 2^{k-L+1} - 1 + 2^k - 1 + \sum_{l=2}^{L-1} 2^{L-l} - 1 \qquad (5.4) \\
= \;& l + 2^{k-L+1} - 1 + 2^k - 1 - (2^{L-1} - 1) - (2^0 - 1) + \sum_{l=1}^{L} 2^{L-l} - 1 \\
= \;& l - l + 2^{k-L+1} - 1 + 2^k - 1 - (2^{L-1} - 1) + \sum_{l=1}^{L} 2^{L-l} \\
= \;& 2^{k-L+1} - 1 + 2^k - 2^{L-1} + \sum_{l=1}^{L} 2^{L-l}
\end{aligned}
$$

$$
\begin{aligned}
&= 2^{k-L+1} - 1 + 2^k - 2^{L-1} + 2^L - 1 \\
&= 2^{k-L+1} + 2^k - 2^{L-1} + 2^L - 2 \\
&= 2^{k-L+1} + 2^{L-1} + 2^k - 2
\end{aligned}
$$

If this value is equal to or greater that 2^{k+1}, (the length of P'), then by Lemma 2, P' does indeed have memory requirements less than $((k+1)+1)\times \mid r_i \mid$. Therefore, for this to be true, for values of L where $2 \leq L \leq k+1$,

$$
\begin{aligned}
& 2^{k-L+1} + 2^{L-1} + 2^k - 2 \geq 2^{k+1} \qquad (5.5) \\
\Rightarrow\ & 2^{k-L+1} + 2^{L-1} \geq 2^{k+1} - 2^k + 2 \\
\Rightarrow\ & 2^{k-L+1} + 2^{L-1} \geq 2^k + 2
\end{aligned}
$$

By Lemma 4, this is a contradiction and, therefore, P' must have memory requirements greater or equal to $((k+1)+1)\times \mid r_i \mid$.

Therefore, for any value of L where $1 \leq L \leq 2^{k+1}$, if the memory requirements of a program of size 2^k are at least $(k+1)\times \mid r_i \mid$, then the memory requirements of a program of size 2^{k+1} are at least $(k+1+1)\times \mid r_i \mid$. Since this holds for $k=1$ from the base case, then it also holds for all values of $k \geq 1$.

·//·

5.6.1 Understanding Where Scoped Ordering Fails

Having shown that some programs will require an infinite amount of memory, it is necessary to understand the pattern of region lifetimes that leads to this occurring. A solution can then be proposed that addresses this problem. Recall that *annotateWithEIA()* first identifies the set S of every region in order to find the set of possible annotations. For any given region r_i, this set identifies a region sequence (called the chain) along which an **EXIT** annotation would be placed for every **ENTER** annotation placed alongside that of r_i. Assume without loss of generality that these **ENTER** annotations are placed at r_i^-. The memory requirements at r_i are equal to the sum of $\mid r_i \mid$ and all those regions between r_i^- and the last **EXIT** annotation in the chain of r_i which are wrapped in an **EIA-ENTER/EIA-EXIT** annotation pair which indexes an **ENTER/EXIT** annotation pair at r_i^-. Therefore, the memory requirements at r_i are infinite if and only if an annotation of an infinite program could require an infinite number of regions to have an allocation context at r_i^-. By observing the routine in Algorithm 8 for *annotateWithEIA()*, an infinite amount of memory would be required if the size of the S-set for r_i is infinitely large. However, if the size

of this set is finite, then the memory requirements at r_i are also finite. Consider the following code fragment from Algorithm 10 which generates the S-set for a region:

```
While{i<=|S|}
{
    S=S+{r_x ; EDP(S[i-1])<r_x<EDP(S[i]) ^ EDP(r_x)>EDP(S[i])}
    SortByEDP(S)
    i++
}
```

The size of the S-set for a region would be infinitely large if this loop never terminates as a new element is added to S at every iteration. Therefore, if it can be proven that a new region is added to the S-set at every iteration of this loop, then the memory requirements of the program are infinite. For example, in the program used in the proof above, it can be shown that the size of the S-set for r_1 is equal to the number of iterations of the loop. When the size of this loop is infinite, then the S-set for r_1 is infinite and the memory requirements for the program are therefore also infinite.

5.6.2 Addressing the Failure of Scope Ordering through Recyclable Memory

As described in Section 2.3.1, a number of approaches have been proposed in the literature to address the problem of infinite memory requirements in scoped programs. This is achieved primarily through mappings of the scoped model to alternative memory models such as the memory pool model. However, such solutions lose the advantages of the scoped approach in guaranteeing the absence of fragmentation. The above investigation exposes a solution that can maintain this guarantee whilst also finitely bounding the memory requirements of all programs: in order to guarantee that finite memory is required, the size of the S-set for all regions must be made finite. An extension to RPM − Scoped that achieves this is described next.

Consider a program P with an infinite loop I which encloses the finite sequence of regions $r_{\overrightarrow{1,n}}$. Let the set of regions for which memory must have been allocated but not deallocated after r_n^+ in the i^{th} iteration of I by the EDP specifications be defined by the set A. Assume that the set A contains only one iteration of a region. Therefore, if r_x is in A, then only one iteration of r_x would not have been deallocated after $(r_n^+, \{I = i\})$. Assume also that at the ki^{th} iteration of I (where $k\varepsilon\mathbb{Z}^+$), the set of regions that would have been allocated but not deallocated by the EDP specification is also A. For example, if $i = 5$, then at the end of the 5^{th}, 10^{th} and 15^{th} execution of r_n^+, the same set of regions would still be allocated in a memory model such as RPM − Unscoped.

By giving special treatment to the regions in A at the chosen value of i, *annotateWithEIA()* can still be used to find the optimal ordering for a finite subset of iterations of the infinite loop. First, the deallocation point of the regions in A are set to $(r_n^+, \{I = ki\})$ and *annotateWithEIA()* is executed on this region sequence. Having found this annotation, the elements in the set A must

be suitably annotated to change their lifetime semantics from that given by *annotateWithEIA()*. In order to achieve this, a new set of memory areas called *recyclable memory areas* is introduced. Each recyclable memory areas is created before the start of the infinite loop and is used as the allocation context for every execution of a region in the set A. The region iterations in A are wrapped in a special annotation pair, **EIA-ENTER*** and **EIA-EXIT***, that indexes its corresponding recyclable memory area. The method for this annotation pair is called *executeInAreaClear()* as the memory area is cleared before it becomes the allocation context of the specified region. A recyclable memory areas is unique in that it is cleared every time the allocation context for a thread is set to a region to which it is associated. Therefore, the lifetime semantics of a region r_x in an **EIA-ENTER*** and **EIA-EXIT*** annotation are that $\text{RAP}(r_x) = E$ and the single element in E is $\text{RAP}(r_x, \{I = ki\}) = (r_x^-, \{I = ki\})$, meaning that the allocation point for r_x at these iterations is at its LAP. This RAP is paired with the single element $\text{RDP}(r_x, \{I = ki\}) = (r_x^+, \{I = I + 1\})$. If the assumption that A contains only one iteration of a region holds, then this specification is correct as the i^{th} iteration of r_x in this annotation pair would have to have a deallocation point specified in its EDP that was before the $i^{th} + 1$ iteration of r_x.

As an example, consider the program used in the previous proof which consists of a single region r_1 in an infinite loop and for which $\text{EDP}(r_1, \{I > 0\}) = (r_1^+, \{I = I + 1\})$. This program is depicted in Figure 5.17 (a). At the i^{th} iteration of r_1, only one element of all iterations of r_1 has been allocated and has a deallocation point after r_1^+ and therefore $A = \{r_1\}$. Assume that the developer chooses i to be 10. The resulting program is given in Figure 5.17 (b). The first nine iterations of r_1 are executed in the context as specified by *annotateWithEIA()*. The tenth iteration however is executed in the memory area created for r_1^* despite the fact that *annotateWithEIA()* would have created another memory area for this iteration of r_1. Therefore, a memory size of zero is assigned for this region iteration when finding an annotation for the chain in *annotateWithEIA()*. The final annotation is as shown in Figure 5.17 (c). Note that once r_1^* has been reached at any iteration of the infinite loop, the only memory still allocated is that for which the allocation context is r_1^*.

The above extension to $\text{RPM} - \text{Scoped}$ makes two assumptions. First, it is assumed that the set A contains only one iteration of a region so that if r_x is in A, then only one iteration of r_x would not have been deallocated after $(r_n^+, \{I = i\})$. This assumption can be removed by allowing as many recyclable memory areas for r_x as is necessary for the chosen value of i. Since i is finite, a finite amount of recyclable memory is required for a finite number of region iterations and, hence, memory requirements are also finite. The second assumption is that at the ki^{th} iteration of I, the set of regions that would have been allocated but not deallocated by the EDP specification is also A. This assumption can also be removed by creating recyclable memory areas for all regions in the possible sets of A. Note that in the worst case, a recyclable memory area is needed for as many

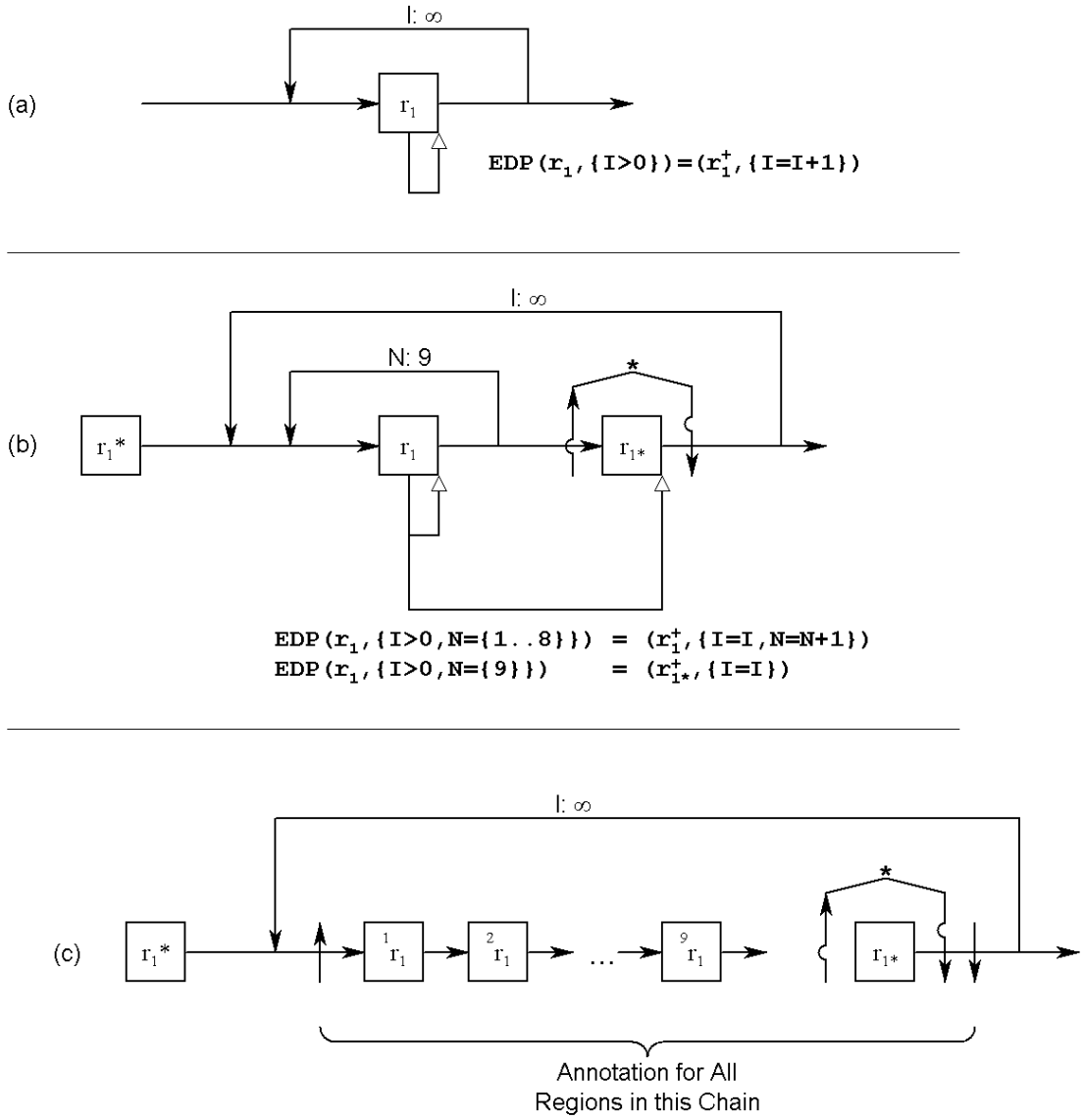


Figure 5.17: Using Recyclable Memory Areas to Bound Memory Requirements

regions as are allocated at the same time executed in an instance of recyclable memory. In this case, the memory requirements would be as large as that experienced in a static model. It is noted in passing that recyclable memory areas could be shared between different regions by using a technique similar to that used for memory sharing for programs in cRPM$^?$ and described in Section 4.3.2. This optimisation is not considered further here.

Finally, two further assumptions were made previously that must be addressed. The first was that no regions prior to the infinite loop in a program could have deallocation points that fall inside the infinite loop. This assumption was made in order that the discussion for recyclable memory would not have to consider regions outside the infinite loop and the value for i could be specified that could be attributed to this loop. In practice, this assumption can be made a requirement by unrolling the infinite loop sufficiently until no deallocation points lie within the infinite loop for regions outside this loop. The second assumption was that every region in the infinite loop I has a single inter-region location specified for all iterations of the loop. This assumption was made only to simplify the discussion above. In practice, an arbitrary number of sets of deallocation points for the infinite number of regions can be specified. As long as the memory requirements are finite in RPM$^{\rightarrow}$, the number of recyclable memory areas required will still be finite for a non-infinite value of i.

5.6.3 Recyclable Memory Areas and the Entropy Hypothesis

It would be incorrect to view recyclable memory as a technique that shifts overheads between space and time within the same memory model. The time overheads are still approximately the same as those that would be experienced in a system with infinite memory as fragmentation is still avoided. Also, it is reasonable to assume that the difference in execution time between entering a region assigned a recyclable memory area and entering a regular region would also be comparable. Nevertheless, the memory requirements are reduced from unbounded to bounded requirements when using recyclable memory. Since the source program and specified lifetimes are the same in both cases, the Entropy Hypothesis would suggest that this reduction in overheads would be achieved if more information is taken advantage of in a model that uses recyclable memory than in one which does not. The constraints set by the scoped model on region lifetimes are the cause of this information loss as these can no longer be expressed exactly and independently of other regions. Rather, the allocation and deallocation points of a region may depend on that of another region. If this relationship is recursive, then an allocation and deallocation point for all regions can not be expressed. As shown above, this is manifested by the S-set of regions being infinite. Recyclable memory is therefore a means to terminate this recursion by allowing an alternative way of expressing the lifetime of a region such that this does not depend on an infinite number of other regions. Although other

solutions such as memory pools achieve the same goal, there is no guarantee in the semantics of this model that fragmentation would not occur. With recyclable memory areas however, this guarantee is maintained.

5.7 Region Sharing and Synchronisation in RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$

Region sharing and synchronisation were considered in Sections 4.2.2 and 4.2.4 for multi-threaded programs where each thread was specified using RPM$^{\rightarrow}$ or cRPM$^{\rightarrow}$. Consider a program $P = \langle T, r_m, H\rangle$ in RPM$^{\|\sigma\rho}$. Recall that T is a set $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$ and each τ_i is a tuple $\tau_i = \langle R_i, <_i, \mathfrak{E}_i\rangle$. As done previously for RPM$^{\rightarrow}$ and RPM$^{\circlearrowleft}$, the goal is once again to derive a program in the constrained version of RPM$^{\|\sigma\rho}$, cRPM$^{\|\sigma\rho}$ that conforms to the constraints of the scoped model. Since the semantics of **ENTER/EXIT** and **EIA-ENTER/EIA-EXIT** annotations as defined so far apply only to a single thread, the lifetime of regions as described through region sharing can not be expressed using the same semantics. There are two possible solutions to address this. The first solution is to extend the semantics of these annotations so that they can capture the lifetime of a region across all the threads in the program. It could be argued that the RTSJ adopts this solution by allowing several regions to execute within the context of a single memory area. However, this functionality does not sufficiently capture the semantics of region sharing as described in RPM$^{\|\sigma\rho}$. This is because if region $r^i_{i'}$ holds a cross-thread deallocation point as specified in EDP($r^i_{i'}$) to the inter-region location before $r^j_{j'}$ (that is, ${r^j_{j'}}^+$) and these regions share the same memory area, then the semantics of the cross-thread deallocation point are only satisfied if τ_j enters the shared space for executing both $r^i_{i'}$ and $r^j_{j'}$ before τ_i. In practice, this argument is more complex due to the single-parent rule of the RTSJ. The second solution, and that opted for in this thesis, is to eliminate from the analysis carried out in *annotateWithEnter()* or *annotateWithEIA()* all regions whose deallocation points depend on the completion of execution of regions that do not lie in the same thread. This is achieved by setting the size of these regions to zero. Note that the inter-region locations before and after these regions still exist in practice and will be taken into consideration by the memory subsystem. The rationale for choosing this approach is that the indeterminacy of the lifetime of regions with cross-thread deallocation points could contribute to fragmentation of memory. As will be described in Section 5.8, the memory subsystem for RPM − Scoped that considers multi-threading will be organised such that memory is partitioned into fragmentation-free and non fragmentation-free areas. Regions with cross-thread deallocation points will therefore be placed in areas that are subject to fragmentation and any resulting overheads. No nested annotations similar to those for *enter()* and *executeInArea()* are required to specify a cross thread deallocation point. Rather, a new method *deallocateAfter*($r^i_{i'}$, $r^j_{j'}$) is introduced that specifies that the memory for

$r^i_{i'}$ can only be deallocated after $r^j_{j'}$ has finished executing. In order to simplify the implementation of an extension to the RTSJ described in Section 6.1.2, if a region $r^{j''}_j$ comes before $r^{j'}_j$ (that is $r^j_{j''} < r^j_{j'}$), it can also be specified that *deallocateAfter*($r^i_{i'}, r^j_{j''}$). Therefore, if $r^{j'}_j \varepsilon \text{EDP}^{\text{cRPM}^{\|\sigma}}(r^{i'}_i)$, then *deallocateAfter*($r^i_{i'}, r^j_{j'}$) must be specified but *deallocateAfter*($r^i_{i'}, r^j_{j''}$) can also be specified if $r^j_{j''} < r^j_{j'}$.

Recall that by exposing synchronisation points through the set H, the worst case memory requirements could be reduced by allowing consideration of only those region sequences that could be executing concurrently in the runtime. This information would be of no benefit when using *annotateWithEnter()* since all regions have only one possible pair of ideal locations for their **ENTER/EXIT** annotations. However, since every region can have several pairs of locations that could prove ideal when using *annotateWithEIA()*, an optimal annotation for individual threads could prove suboptimal when synchronisation is considered for $\text{RPM}^{\|\sigma\rho}$ programs. For example, consider a program with two threads as shown in Figure 5.18 (a) with memory requirements as depicted by the number next to each region. Given the deallocation points as shown here, two possible annotations for τ_1 could be derived using *annotateWithEIA()* for which the memory requirements at each region are shown in Figure 5.18 (b) and (c). Although the memory requirements in the worst case are 30 using either annotation, the memory requirements of the whole program are equal to $30 + 25 = 55$ in (b) but only $30 + 1 = 31$ in (c). Since deriving an optimal solution for the whole program would require considering every possible annotation against the set of execution traces, such a solution does not scale. However, non-optimal heuristics could be employed whereby a set of candidate annotations are derived from *annotateWithEIA()* rather than a single annotation for a thread. The memory requirements of the cross-product of these set of annotations when considering the set of possible execution traces would be found in order that a single annotation for each thread can be chosen. A new method *synchroniseWith*($r^{i'}_i, r^{j'}_j$) is introduced that specifies that $r^{i'}_i$ can not begin executing until after $r^{j'}_j$ is finished executing. Therefore, if this is specified, then the pair $(r^{i'}_i, r^{j'}_j) \varepsilon H$ in the program definition of $\text{cRPM}^{\|\sigma\rho}$.

5.8 Memory Subsystems for RPM – Scoped

A memory subsystem for RPM – Scoped must take advantage of the information expressed in the Spatio-Temporal Specification for a program. There are three items of information expressed in this case:

1. the lifetime of regions as expressed through the annotations of *enter()*, *executionInArea()* and *executeInAreaClear()*,

2. the lifetime of regions expressed in cross-thread EDPs in programs that have region sharing,

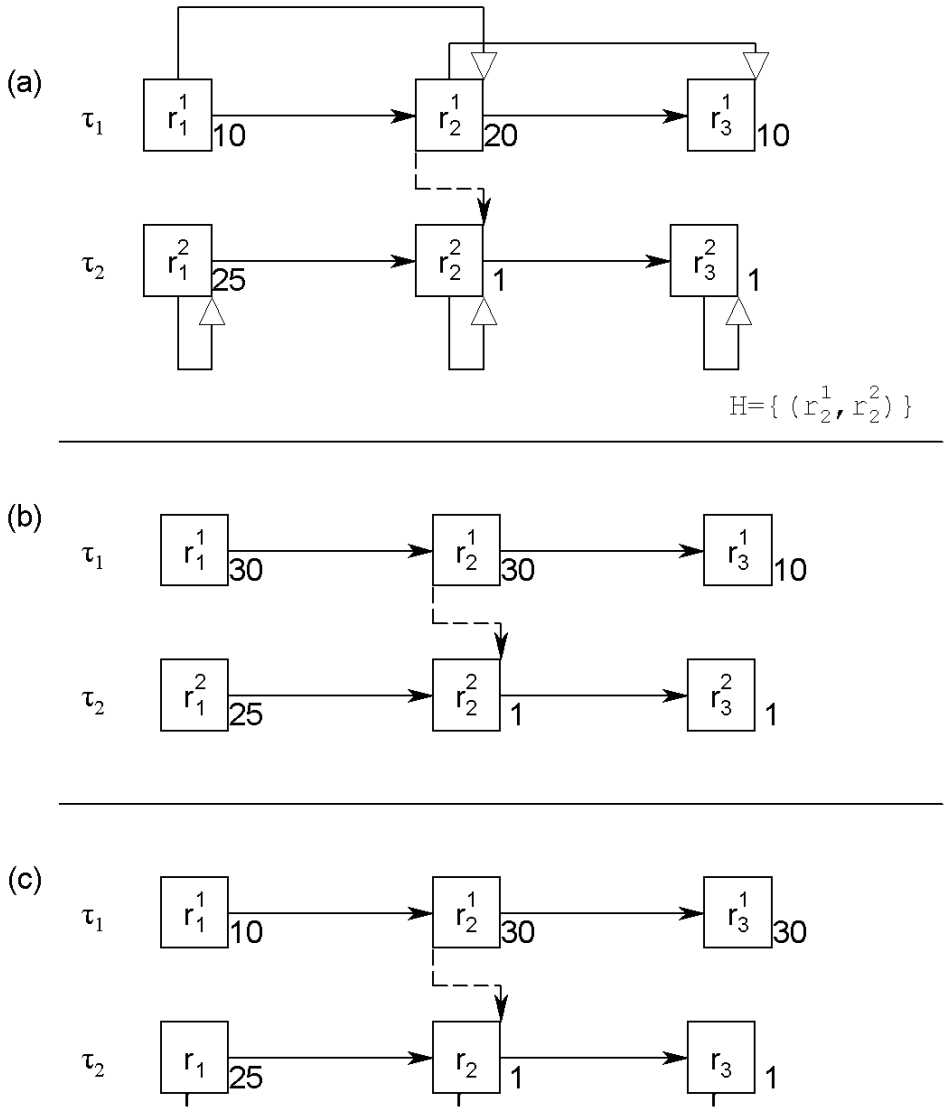


Figure 5.18: Finding Optimal Annotations for Programs in RPM – Scoped$^{\mathrm{cRPM}^{\|\sigma\rho}}$

and

3. the synchronisation points specified in H.

For a program $P = \langle R, <, \mathfrak{E}, \mathfrak{A}, \mathfrak{D} \rangle$ in cRPM$^{\rightarrow}$, a memory subsystem SMM$^{\rightarrow}$ is proposed that takes advantage of the first of these three items of information by allocating and deallocating memory as specified in the RAP and RDP specifications in $\mathfrak{A}$ and $\mathfrak{D}$. Since the order of allocation and deallocation is such that no fragmentation can ever occur, an implementation of SMM$^{\rightarrow}$ need only recall the position at which free memory starts. In contrast to EMM − ONLINE which required a DMA with an allocation and deallocation policy (and possibly a defragmentation routine), this overhead is minimal and both the space and time requirements are less than those experienced in EMM − ONLINE for programs with large variances in the size of regions. In contrast to EMM − OFFLINE however, the additional constraints of a scope ordering result in greater memory requirements. A more detailed investigation on the implications of this observation is carried out in Section 7.2. Programs in cRPM$^{\circlearrowleft}$ without conditional loops can also use SMM$^{\rightarrow}$ with the execution engine passing only the relevant allocation and deallocation calls that apply to the currently executing loop iterations. Similarly, programs in cRPM$^{?}$ can use SMM$^{\rightarrow}$ for which all allocation calls reached at an **ENTER** annotation are passed to SMM$^{\rightarrow}$ (as allocation is independent of the chosen path) and deallocation calls are passed only if the path that was taken is that specified in the additional annotations attributed for **EXIT** annotations that are reached. Note that although memory sharing can be adopted as for any cRPM$^{?}$ program, it is assumed that the implementation of this logic is defined at a higher level than SMM$^{\rightarrow}$. Recyclable memory is created on an **ENTER** annotation just as any other region. However, the memory for these regions is cleared by SMM$^{\rightarrow}$ on an **ENTER-EIA*** being reached. A memory map for SMM$^{\rightarrow}$ for programs in cRPM$^{\rightarrow}$, cRPM$^{?}$ and cRPM$^{\circlearrowleft}$ is given in Figure 5.19 (a). Note that in the example shown here, r_a and r_b are regions for which memory is never deallocated.

For a program $P = \langle T, r_m \rangle$ in cRPM$^{\|\sigma}$, a memory subsystem SMM$^{\|\sigma}$ is proposed that takes advantage of the first two items of information listed above. For regions with no cross-thread deallocation points, an identical approach to SMM$^{\rightarrow}$ can be adopted where every thread in T has memory reserved for it that is equal to the worst case space derived using *annotateWithEIA()* on that thread. As described in Section 5.7, those regions with cross-thread deallocation points are considered to have zero size. The memory for these regions is allocated in a special memory area which uses a fragmentation DMA similar to that used by EMM − ONLINE. This is necessary as the exact order of deallocation of these object often can not be determined. Therefore, an ordering similar to that derived using *annotateWithEIA()* that guarantees no fragmentation will occur can

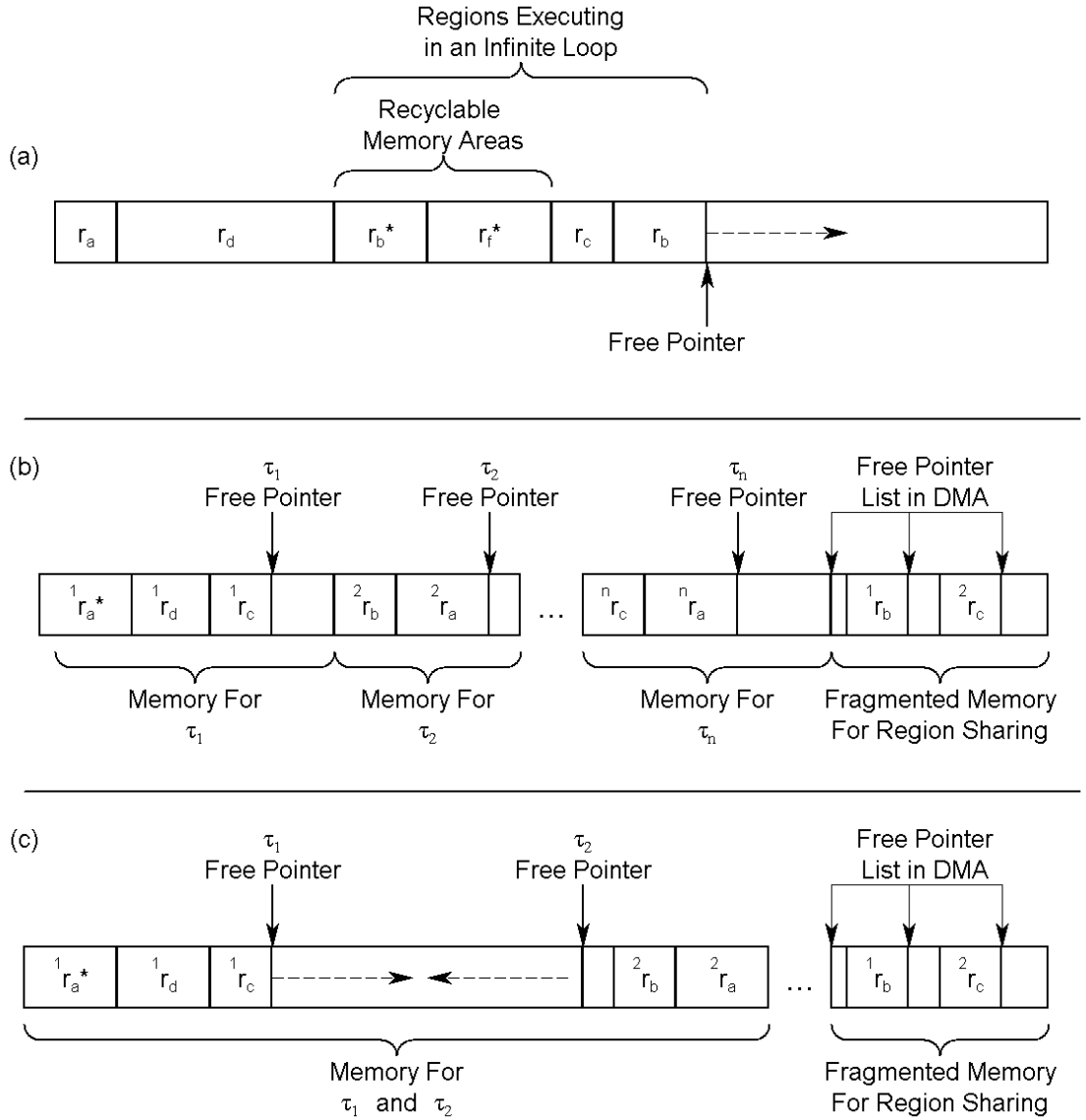


Figure 5.19: Memory Maps for SMM$^{\rightarrow}$, SMM$^{\|\sigma}$ and SMM$^{\|\sigma\rho}$ Respectively

not be specified.[9] As is conformal with the Entropy Hypothesis, this indeterminacy incurs additional overheads that are manifested in this case in the possibility of fragmentation occurring in the memory space. As in the case of EMM − OFFLINE, the benefits of expressed region sharing are partially lost in this memory subsystem. A memory map for SMM$^{\|\sigma}$ for programs in cRPM$^{\|\sigma}$ is given in Figure 5.19 (b). Note that recyclable memory can also be used in SMM$^{\|\sigma}$ on a per-thread basis.

For a program $P = \langle T, r_m, H \rangle$ in cRPM$^{\|\sigma\rho}$, a memory subsystem SMM$^{\|\sigma\rho}$ is proposed that takes advantage of all three items of information listed above. The one-dimensional nature of physical memory means that it is non trivial to carry the advantages of exposing synchronisation down to the memory subsystem while maintaining the advantages of non fragmentation. As shown in the memory map for SMM$^{\|\sigma\rho}$ in Figure 5.19 (b), the simplest solution is to partition memory such that the memory stacks for two threads grow towards each other. As shown in Section 4.2.4, the runtime memory requirements of the program are reduced to those experienced in the reachable execution traces of these two threads. However, the reduced memory requirements achieved through the synchronisation specification of these two threads with other threads in the system is lost. Although alternative memory maps that allow for synchronisation between a greater number of threads could be derived, this investigation is beyond the scope of this thesis. Showing that synchronisation between pairs of threads can be taken advantage of right down to the level of the memory subsystem suffices here. This is an important observation because, as noted in Section 4.4.3, there is no known way of leveraging this information when using existing offline strategies.

Finally, the memory requirements given by $\text{MemReq}^{\text{RPM}}_{\text{SMM}^{\|\sigma\rho}}(P)$ can be derived. $\text{MemReq}_{\text{SMM}^{\|\sigma\rho}}(P)$ which would have to be added to $\text{MemReq}^{\text{RPM}}(P)$ to give the total memory requirements $\text{MemReq}^{\text{RPM}}_{\text{SMM}^{\|\sigma\rho}}(P)$. Using the memory map described, $\text{MemReq}_{\text{SMM}^{\|\sigma\rho}}(P)$ incurs the cost of not being able to make full use of the expressed synchronisation. Therefore, in calculating $\text{MemReq}_{\text{SMM}^{\|\sigma\rho}}(P)$ it is necessary to start with a value for $\text{MemReq}^{\text{RPM}}(P)$ that only considers the synchronisation elements for which the pairs in H identify regions that are to share the same space. As with EMM − ONLINE, three additional overheads of SMM$^{\|\sigma\rho}$ and its DMA must be considered: the overheads of any information stored by SMM$^{\|\sigma\rho}$; the overheads of the book-keeping structures of the DMA for fragmenting and non-fragmenting areas; and the extra space required due to fragmentation in the fragmenting area. The space overheads of SMM$^{\|\sigma\rho}$ would include information such as the STS and a bitmap similar to that described for EMM − ONLINE and that is used to identify when the memory for regions in the fragmenting memory space can be deallocated. Unlike EMM − ONLINE, SMM$^{\|\sigma\rho}$ incurs no space overheads due to fragmentation in the non-fragmenting areas of scopes, even though the overall memory requirements could be

[9] In practice, there may be occasions when using synchronisation when such an ordering could be derived. These regions could therefore be defined as another thread in P with suitable synchronisation points. The number of regions in the area subject to fragmentation is therefore reduced.

greater than those in EMM − OFFLINE. The worst case overheads for the fragmenting areas must however be derived using traditional analysis. As noted above, the overheads of the book-keeping structures of the DMA are minimal for non-fragmenting areas. However, for the fragmenting areas, these are equal to those specified for the chosen DMA implementation. The timing characteristics of EMM − ONLINE $\text{TimeReq}^{\text{RPM}}_{\text{SMM}^{\|\sigma\rho}(P)}$ are specified for regions for which memory is created in fragmentation-free areas and for regions for which memory is created in the fragmenting space. Whereas the latter would be equivalent to the time overheads experienced in EMM − ONLINE, the time overheads in the former are equal only to a pointer update on allocation and deallocation. This is comparable with the overhead of EMM − ONLINE which must simply read the area in memory to carry out an allocation. Finally, the additional cost of switching between allocation contexts must also be considered. This overhead is minimal as context changes can be implemented with just a pointer update to the current allocation context on *enter()*, *executeInArea()* and *executeInAreaClear()* being invoked.

5.9 Summary

In this chapter, the memory model RPM − Scoped was introduced. This memory model expresses a scoped model that is achieved by using the STS of a program in three versions of the RPM ($\text{RPM}^{\rightarrow}$, $\text{RPM}^{?}$ and a subset of $\text{RPM}^{\circlearrowleft}$) to derive another STS that conforms to the constraints of a scoped model. Algorithms that carry out this transformation automatically were considered in each case. The second part of the thesis hypothesis was proven by showing that the memory requirements of a program in $\text{RPM}^{\circlearrowleft}$ could be infinite in $\text{RPM} - \text{Scoped}^{\text{cRPM}^{\circlearrowleft}}$ where they would be finite in $\text{RPM} - \text{Unscoped}^{\text{RPM}^{\circlearrowleft}}$. A solution to this problem was proposed that makes use of recyclable memory for which context changes for regions that use this memory carry unique semantics. Multi-threaded programs and the implications of region sharing and synchronisation as specified in $\text{RPM}^{\|\sigma\rho}$ in a scoped model were investigated. Finally, memory subsystems for RPM − Scoped were proposed that leverage the advantages of the information expressed in these different versions of the RPM. It was shown that, as with existing offline algorithms, the indeterminacy of region sharing is manifested as overheads due to fragmentation at runtime. However, although taking full advantage of expressed synchronisation was shown to be non-trivial, a partial benefit can be achieved with even trivial allocation strategies where this could not be done using existing offline allocation algorithms.

Chapter 6

Applying the Region Partitioning Model to the RTSJ

In Chapter 5, programs specified in unconstrained versions of the Region Partitioning Model (RPM) were used to describe the relative ordering of regions on which the constraints of a scoped model could be applied. Memory subsystems for RPM − Scoped that take advantage of the information expressed in these programs were developed. The focus is now shifted back to the RTSJ and, in particular, to the adaptation of the RTSJ for applications that target RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$. This chapter begins with an investigation of how region lifetimes are specified in the RTSJ. A severe restriction of its scoped memory model is addressed by means of an extension that allows more flexibility in the way that the lifetimes of regions are specified. This extension is described in detail in Section 6.1 within the context of RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$. The constrained and unconstrained versions of the RPM are abstract models that have to be implemented in a development environment. In particular, the specification of programs and associated allocation points, deallocation points and synchronisation points of regions must be specified in that environment. Section 6.2 demonstrates how this is achieved in a Java environment for RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ programs by developing Java classes to express this information. A prototype implementation of a runtime based on RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ is also developed in this section in order to execute these programs on top of the RTSJ memory model. Finally, the differences between RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ and the RTSJ memory model are investigated. For example, regions in the RTSJ can be shared between threads where this is not allowed in RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ (or any of the RPM − Scoped models). The advantages and disadvantages of having this different functionality are investigated in Section 6.3.

6.1 Separating Region Lifetime from the Reference Graph

All the different versions of the RPM require that every region have a specified lifetime provided in the EDP specification of that region. The techniques used to identify this information remained

unspecified but could be derived by a number of approaches. As shall be demonstrated in Section 7.1, one approach is to immediately direct development towards targeting the RPM by considering each region as an independent process of the global program for which the dependency of each process on other processes is known. Another approach is to develop the application as if targeting an automatic, fine grain memory model and then to derive these dependencies based on an exploration of the object graph. This is the solution adopted in [55] and [62] where objects are assigned to regions based on the reference graph of the program in order to satisfy the RTSJ scoped reference rules. Equating the lifetime of objects to the reference graph rather than implicit information known about their real lifetimes results in imprecision in the expression of memory usage in the memory model. As a direct manifestation of the Entropy Hypothesis, it can be shown that the runtime memory requirements of a program could be increased. As an example of this, consider the following Java program fragment:

```
public static void main(String[] args) {
      ClassA a;
      ClassB b;
      a = new ClassA();
      do
      {
          b = new ClassB();
          a.field_to_ClassB = b;
          a.field_to_ClassB = null;
          b = null;
      }
      while (true);
  }
```

The program first creates an object of type `ClassA` called a. An infinite loop is then entered in which a new object of type `ClassB` called b is created and a field for object a is set to reference b. As the object b is no longer needed, all references to this object are set to `null`. In an environment which uses a garbage collector, b could be marked for finalisation and deallocated at this stage. Using region-based notation for programs in RPM$^{\circlearrowleft}$, the lifetime of the objects created in this program would be given by $\text{EDP}(a) = \{\text{EDP}(a, \{\}) = (\infty^+, \{\})\}$ and $\text{EDP}(b) = \{\text{EDP}(b, \{L > 0\}) = (b^+, \{L = L\})\}$ and the memory requirements are given by $\text{MemReq}^{\mathcal{M}^{\text{RPM}^{\circlearrowleft}}} =\mid a \mid + \mid b \mid$. If the lifetime of objects is dependent on the object graph, then it will be required that when an object a holds a reference to an object b, then b can not be deallocated before a, even if this reference is removed as in this example. This requirement is enforced in the RTSJ through the scoped memory reference rules described in Section 2.2. Therefore, the memory for all instances of `ClassB` would never be deallocated as a is never deallocated.

The motivation for the RTSJ reference rules is to maintain program safety by guaranteeing that dangling pointers can never occur. As shown in this example, combining the lifetime of regions with

the reference graph leads to pessimistic or, in some cases, unbounded memory requirements. In order to address this problem, an extension to the RTSJ is proposed that separates the lifetime of regions from the object graph. However, it is not sufficient to simply remove the RTSJ reference rules by citing an assumption that the expression of region lifetime is known to be correct. This extension must therefore provide this separation while defining deterministic failure semantics when a dangling pointer is followed. The remainder of this section proposes and evaluates such an extension that is based on Java's reference objects mechanism. This mechanism is described in the package `java.lang.ref` and defines classes that allow the programmer a limited degree of interaction with the garbage collector [3]. In particular, three classes are defined that allow the creation of *reference objects*. A program may use a reference object to maintain a reference to some other object in such a way that the latter object may still be garbage collected. A program may also arrange to be notified some time after the collector has determined that the reachability of a given object has changed. Similar functionality is integrated into $\mathrm{RPM} - \mathrm{Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$ in order to provide the necessary failure semantics for objects that invoke or follow references to other objects in regions that may have already been deallocated. Note that this extension is also relevant to the RTSJ and a prototype implementation has been developed by extending the RTSJ [35].

6.1.1 Using Reference Objects in Java

The assignment of an existing or new object in Java creates a reference called a *Strong Reference*. A strongly referenced object can not be garbage collected, that is marked for finalisation, finalised and its memory reclaimed. The three reference objects provided in the `java.lang.ref` package are *Soft References*, *Weak References* and *Phantom References* and are created by instantiation of the `SoftReference`, `WeakReference` and `PhantomReference` classes respectively. The constructors for reference object classes take a reference to some object as a parameter for which the required reference is created. For example in the code:

```
Object x = new Object();

WeakReference w = new WeakReference(x);
```

`x` strongly references the newly created object while `w` is a weak reference to that object. The object referred to by a reference object is called the *referent*. Here, `x` is the referent of `w`. The reference object classes extend the abstract class `Reference` that provides the basic functionality for all reference types. These objects contain a field called `referent` that provides the hook used by the application into the garbage collector. Assignments to this field are treated specially by the garbage collector. A set of methods in this package provide the interface for working with reference objects. The *get()* method returns a (strong) reference to the referent if the object has not already

been marked for finalisation by the garbage collector, and `null` otherwise.[1] The *clear()* method clears the reference object; that is it sets the `referent` field to `null`. When any reference object is cleared, *get()* returns `null` for all reference object types. The garbage collector may free the memory occupied by referents once all the associated reference objects are cleared and if the object is not strongly reachable. Note that reference objects are immutable, that is there is no *set()* method to set the `referent` field.

Reference objects allow for some limited degree of interaction with the garbage collector in two ways. Firstly, whereas objects that are strongly referenced are never finalised as long as there remains at least one strong reference in the heap, reference objects do not prevent their referents from being collected. Secondly, a referent object may register itself with a `ReferenceQueue` onto which it is enqueued when its reachability changes. Each reference object encapsulates different semantics that define when: (1) the referent of a reference object can be marked for finalisation, (2) the reference object is cleared, (3) the reference object is enqueued on a registered reference queue and (4) the referent's memory is reclaimed.

The RTSJ does not specify the semantics for reference objects when used with objects in a scoped memory area. An investigation into these semantics and how these would be implemented in a real-time environment is carried out in Appendix D. As this is a contribution that falls outside the scope of the RPM, it will not be investigated further here. A new type of reference object is proposed next that captures the semantics of references between regions for which the lifetime of objects is not dependant on the reference graph as is currently the case in the RTSJ. This reference object is called a *Scoped Reference Object* and allows references to break RTSJ scoping rules while providing deterministic failure semantics. Although the semantics of these new reference types are only considered for RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$, a similar approach could also be adopted for other RPM LSLs.

6.1.2 Scoped Reference Objects in RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$

Scoped Reference Objects provide a mechanism by which references from objects in a region r_1 to objects in a region r_2 where r_1 outlives (or may outlive) r_2 are legal. In the single-threaded versions of the RPM, this means that the memory for r_1 could be placed by *annotateWithEIA()* in a less deeply nested location in the memory stack than that for r_2. In multi-threaded versions of the RPM, this could be due to a cross-thread reference between $r^i_{i'}$ and $r^j_{j'}$. For the special case of RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$, this would be when *deallocateAfter*$(r^j_{j'}, r^i_{i'})$ is specified and $r^i_{i'}$ has finished executing (and there are no other instances of *deallocateAfter*$(r^j_{j'}, r^i_x)$ specified where *deallocateAfter*$(r^j_{j'}, r^i_x)$ still needs to execute). Scoped reference objects provide a wrapper around

[1] The exception is the phantom reference type as these always return `null` from *get()*.

regular reference semantics that capture the additional failure semantics required when an invocation of a deallocated object occurs. The semantics defined for scoped reference objects are such that the existence of the referent can be guaranteed throughout the invocation if the referent existed at the point of invocation. The class structure for scoped reference objects is developed next. This is followed by the specification of the semantics of these references between objects in regions in the same thread and objects in regions in different threads. Finally, the reference rules for $\mathrm{RPM} - \mathrm{Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$ are summarised.

The `ScopedObject` and `ScopedReference` Classes

A new tagging interface called `ScopedObject` is introduced whereby instances of classes that implement this interface may be created in scoped memory and be referenced by special scoped reference objects. Scoped reference objects are proxy objects which hold a reference to the scoped object (the referent). The class `ScopedReference` is introduced that extends `Reference` and contains the static method *getProxy()* that returns such a proxy object. The application invokes methods of the referent by invoking the methods on the proxy. If the virtual machine deems that the invocation has failed as the memory area of the referent has already been deallocated or could be deallocated during the invocation, then an instance of `ScopedAccessException` is thrown in the proxy. Using a proxy makes *get()* redundant and it is therefore overridden in `ScopedReference` to return `null`. Invoking *clear()* on a scoped reference object makes any subsequent invocation on the proxy fail. As an example, the class hierarchy created for scoped reference objects of the class `SimpleObject` is depicted in Figure 6.1. Consider a class `SimpleObject` that contains the methods *int add(int x, int y)* and *Integer add(Integer x, Integer y)* that return the sum of x and y. If an instance of `SimpleObject` is to be referenced by a scoped reference object, then a new interface `SimpleObjectInterface` that extends `ScopedObject` must be defined that describes all methods in `SimpleObject` that can be invoked from the reference object. Each method must throw the checked exception `ScopedAccessException`. A proxy generator is used to generate proxy classes that are by convention given the name of the implementing class concatenated with the string "Proxy". Hence, the proxy generator in this case generates a class `SimpleObjectProxy`. `SimpleObjectProxy` is made to extend `ScopedReference` and implement `SimpleObjectInterface` and is therefore type compatible with the referent. Figure 6.2 shows how references are depicted in the memory stack when using scoped reference objects to reference an instance of `SimpleObject`.

When *getProxy()* is called by the application, the referent of type `SimpleObject` is passed as a parameter in the call. The result is an instance of `SimpleObjectProxy`. An invocation on this proxy calls the *invoke()* method in `ScopedReferenceObject` that attempts to invoke the associated method of the referent. The virtual machine is consulted to ensure that the invocation can take place. If the scoped memory region holding the object still exists and is legal by the rules defined shortly, then

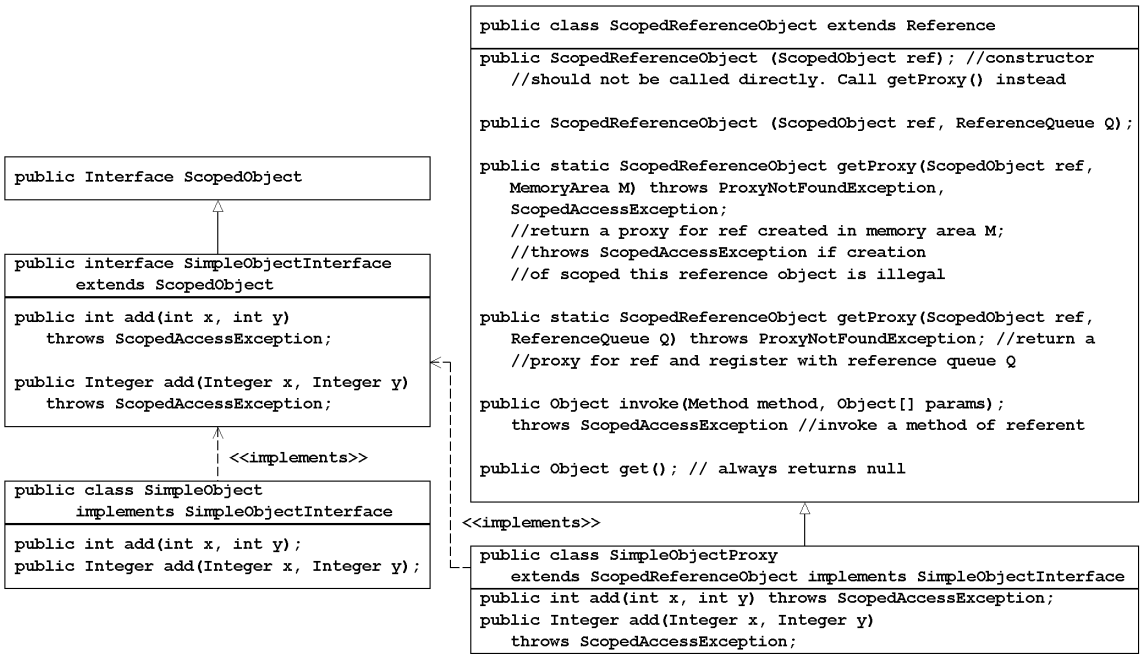


Figure 6.1: Class Hierarchy for Scoped Reference Objects

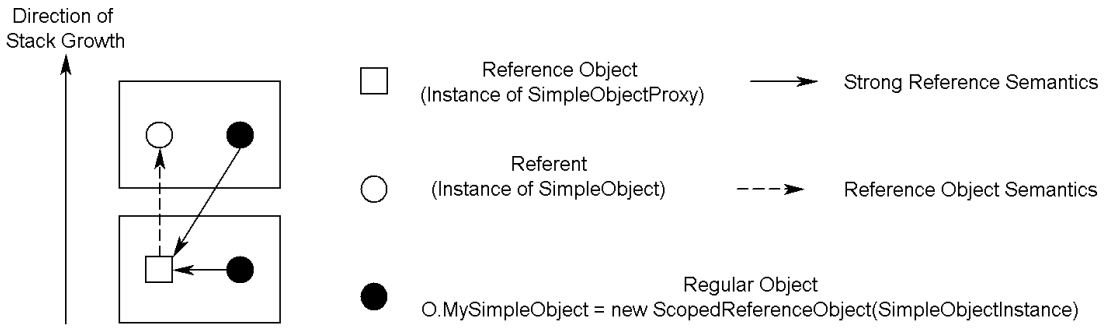


Figure 6.2: Scoped Reference Objects with Referents in the Same Thread

the invocation is carried out with the semantics described below for references in the same thread and referents in other threads. In any case, if the invocation fails, then a `ScopedAccessException` is thrown. Note that the current allocation context during the invocation is that of the reference object and not of the referent. This is consistent with the RPM and RTSJ definitions.

Just as other types of reference objects, scoped reference objects may be registered with reference queues and appended to the queue at some point after the referent is deallocated. This might be useful in applications that require that this occurrence trigger some specified behaviour. This functionality is outside the scope of this discussion and is described further in Section D.3 in Appendix D.

Scoped Reference Objects Between Objects in Regions in the Same Thread

When invoking a scoped reference object whose referent is in a region in the same thread,[2] the referent will exist for the duration of the invocation if the referent existed at the time of invocation. This is because the memory for a region in the current memory stack can only be deallocated at an inter-region location that is reached by that thread. For example, in Figure 6.2, if the filled circle in the least deeply nested memory area represents an object `O` with field `O.MySimpleObject` holding a reference to an instance of `SimpleObjectProxy`, then when invoking `O.MySimpleObject`.*add(1,2)*, the invocation will succeed if `SimpleObject` is still allocated. Moreover, if this method invocation does succeed, then `SimpleObject` will still be allocated when the invocation returns. An implementation of a method for a scoped reference object is therefore as trivial as checking whether the referent has been cleared. If it has not been cleared, then the specified method in the referent is invoked; if it has been cleared, then a `ScopedAccessException` is thrown.

Scoped Reference Objects Between Objects in Regions in Different Threads in $\mathrm{RPM} - \mathrm{Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$

Consider first a scoped reference object in a region $r^i_{i'}$ of thread τ_i for which the referent is in a region $r^j_{j'}$ in thread τ_j and for which *deallocateAfter*$(r^j_{j'}, r^i_{i'})$ is not specified. When invoking a method of the reference object, the existence of the referent is not guaranteed to last for the duration of the invocation. This is because the lifetime of $r^j_{j'}$ is defined by the execution flow of thread τ_j which could attempt to deallocate this region (and hence the referent) at any time. There are at least three possible semantics that could be defined in this case:

1. The first possible semantics are that once an invocation begins, the region holding that object can not be deallocated but the progress of τ_j is not stopped and the memory for outer regions of $r^j_{j'}$ can be deallocated. This approach introduces indeterminacy in the lifetime of region $r^j_{j'}$ with the result that it can not be placed in the fragmentation-free memory area for τ_j.

[2]The definition of the term "in the same thread" is that used for describing regions in the RPM.

Moreover, since the referent may have references to other objects (and recursively to other objects), all these objects would have to themselves be referenced through scoped reference objects and have non-deterministic lifetimes.

2. A second possibility is to halt the progress of τ_j so that the memory for $r^j_{j'}$ is deallocated only once no other threads are invoking scoped reference objects with referents in this region. This solution shifts indeterminacy from the spatial to the temporal domain. Although this may be a suitable solution for time-insensitive systems, it is unsuitable in the real-time domain.

3. A third solution would be to allow an invocation to begin if the referent exists and allow it to fail at any point if τ_j reaches the deallocation point for $r^j_{j'}$. Therefore, if the runtime determines that τ_i is invoking the referent of a scoped reference object in a region that τ_j is attempting to deallocate, then thread τ_i would be interrupted and control transferred to the place where the invocation on the reference object was made so that the checked exception `ScopedAccessException` can be thrown. After this has been done, the memory for τ_j could be deallocated. As arbitrary asynchronous transfer of control has the disadvantage of leaving the system in an inconsistent state, the RTSJ allows the developer to specify where this is allowed to occur by declaring that a method can be asynchronously interrupted. The logic that carries out the invocation of the `ScopedReferenceObject` instance could therefore specify that it can be asynchronously interrupted. However, since this transfer of control only occurs once control returns back to the method carrying out the cross-thread invocation logic, the invocation on the referent would have completed at this point. This approach is therefore tantamount to the second solution above since the memory for region $r^j_{j'}$ can not be deallocated until control returns to the logic carrying out the invocation on the referent.

The inappropriateness of all three possible approaches described here means that scoped reference objects where the proxy is in region $r^i_{i'}$ and the referent is in region $r^j_{j'}$ is disallowed if *deallocateAfter*$(r^j_{j'}, r^i_{i'})$ is not specified. These semantics are consistent with the specified lifetimes declared in the STS of RPM $-$ Scoped$^{\mathrm{cRPM}\|\sigma\rho}$ programs. This point will be discussed further when scoped reference objects are evaluated below.

Consider two objects in regions $r^i_{i'}$ and $r^j_{j'}$ respectively for which *deallocateAfter*$(r^j_{j'}, r^i_{i'})$ is specified. If the object in $r^i_{i'}$ needs to hold a reference to that in $r^j_{j'}$, then a scoped reference object is required for $r^j_{j'}$ but is allowed only if *deallocateAfter*$(r^j_{j'}, r^i_{i'})$ is specified.[3] When $r^i_{i'}$ finishes executing, the scoped reference object is cleared if *deallocateAfter*$(r^j_{j'}, r^i_{i''})$ is not specified for some region

[3]It could be argued that scoped reference objects between regions $r^i_{i'}$ and $r^j_{j'}$ should be allowed if *deallocateAfter*$(r^j_{j'}, r^i_{i''})$ is specified and $r^i_{i'}$ is executed before $r^i_{i''}$. This is because *deallocateAfter*$(r^j_{j'}, r^i_{i'})$ is implied in this case. However, this would increase the complexity of the runtime implementation, particularly in programs with conditionals. This is the reason why *deallocateAfter()* was specified as so in Section 5.7.

$r^i_{i''}$ that executes after $r^i_{i'}$. This arrangement is depicted in Figure 6.3 (a). It is emphasised that the clearing of the reference object occurs when $r^i_{i'}$ finishes executing, not when it is deallocated. Should it be required that the scoped reference object not be cleared until a subsequent region execution $r^i_{i''}$ in τ_i, then *deallocateAfter*$(r^j_{j'}, r^i_{i''})$ would also have to be specified. This arrangement is depicted in Figure 6.3 (b). This approach provides for a simple implementation as the runtime need only maintain a list associated with every region $r^j_{j'}$ of all scoped reference objects with referents in $r^j_{j'}$ for every thread in the program. Once all regions r^i_x for which *deallocateAfter*$(r^j_{j'}, r^i_x)$ is specified finish executing, all the scoped reference objects in τ_i with referents in $r^j_{j'}$ are cleared. Note that this does not imply that the memory for $r^j_{j'}$ is deallocated at this point but only that τ_i can no longer invoke methods in this region. The exact deallocation point for $r^j_{j'}$ will depend on that specified for this region in other threads (including τ_j) as described in Section 5.7. Therefore, as is the case for scoped reference objects with referents in regions in the same thread, an invocation of a scoped reference object is guaranteed to complete if the referent existed when the invocation was made.

Finally, the semantics of invocations in regions that originate from a cross-thread invocation are defined. Consider the case where *deallocateAfter*$(r^j_{j'}, \cdot)$ is specified and an invocation of a scoped reference object with a referent in $r^j_{j'}$ is made by some thread τ_i. Consider further that the invocation on this referent leads to the invocation of another object in some region $r^j_{j''}$. Since the deallocation point of regions for which objects in $r^j_{j'}$ hold references may not be determinable, a similar problem to that described above could occur whereby the memory for a region is deallocated when an invocation is taking place on an object of that region. In order to avoid this, these invocations are disallowed. Therefore, τ_i executing region $r^i_{i'}$ can only ever invoke an object in a region $r^j_{j'}$ if *deallocateAfter*$(r^j_{j'}, r^i_{i'})$ is specified, even if τ_i has an invocation of an object in another region of τ_j on it invocation stack. This arrangement is depicted in Figure 6.3 (c). In order that a checked exception be thrown in this case, it is required that all objects in a region $r^j_{j'}$ for which *deallocateAfter*$(r^j_{j'}, \cdot)$ is specified hold strong references to other objects only if these are also in $r^j_{j'}$. All other references, including those to objects in other regions of τ_j, must be scoped reference objects.

Summarising the Reference Rules for $\mathrm{RPM-Scoped}^{\mathrm{cRPM}^{\parallel\sigma\rho}}$ Programs

In summary, the reference rules for $\mathrm{RPM-Scoped}^{\mathrm{cRPM}^{\parallel\sigma\rho}}$ Programs are as follows:

1. All strong references maintain the same rules as in the RTSJ except in the case of (3) below.
2. A scoped reference object can be created with the proxy object in $r^i_{i'}$ and the referent in $r^j_{j'}$ if:
   (a) $i = j$, or
   (b) $i \neq j$ and *deallocateAfter*$(r^j_{j'}, r^i_{i'})$ is specified.

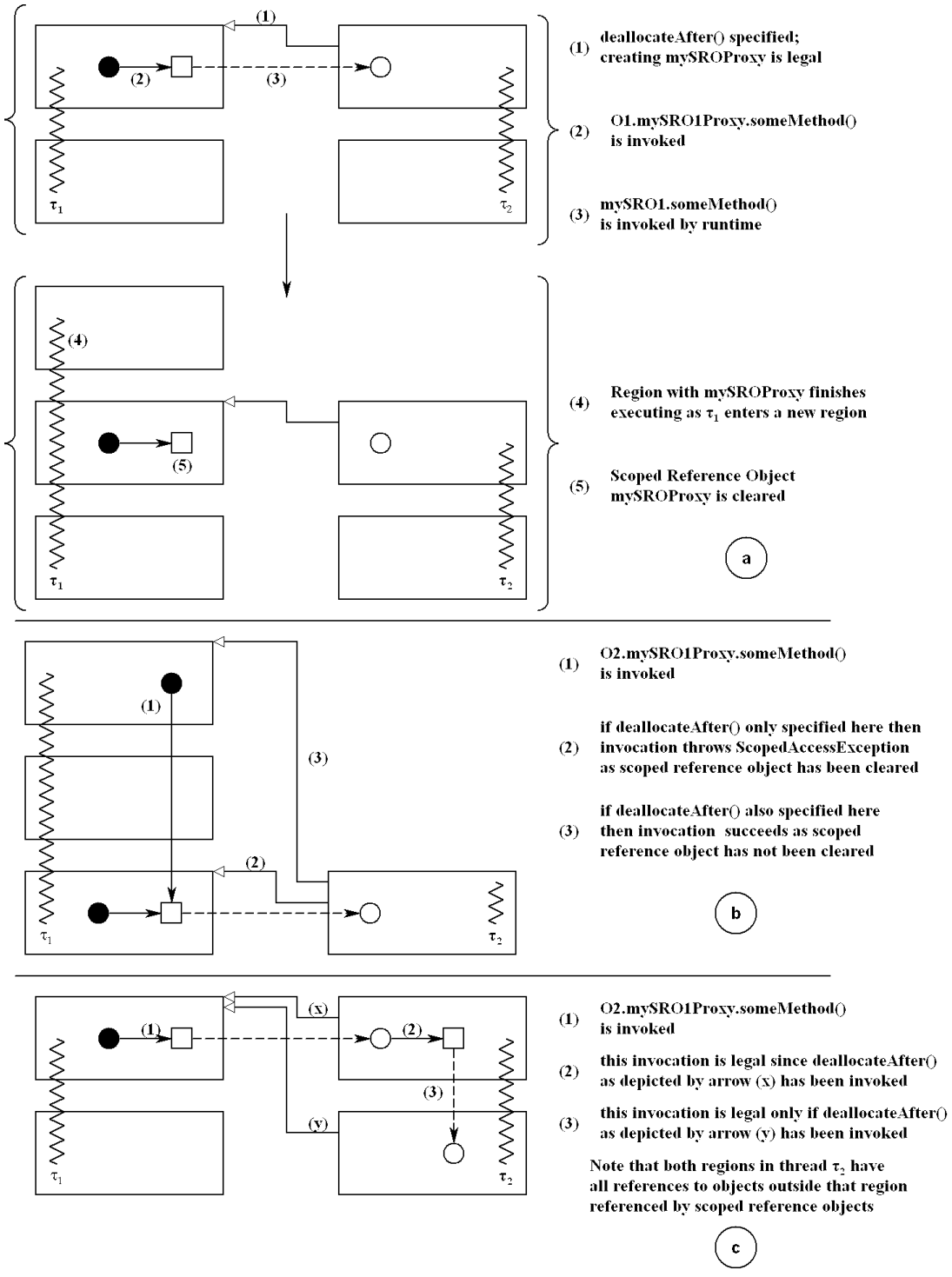


Figure 6.3: Scoped Reference Objects with Referents in a Different Thread

If both of these conditions are not satisfied, `ScopedAccessException` is thrown by *getProxy()*.

3. If *deallocateAfter*($r^j_{j'}, \cdot$) is specified, then any reference from an object in $r^j_{j'}$ to any object in $r^k_{k'}$ where $j \neq k$ (the referent is in a different thread) or $j' \neq k'$ (if the referent is in the same thread, then it is in a different region) must be via a scoped reference object (and must respect item (2) above). Any attempt to create a strong reference when these conditions are satisfied throws an `IllegalAssignmentError` error. Therefore, strong references are only allowed between objects in $r^j_{j'}$ and not between $r^j_{j'}$ and other regions when *deallocateAfter*($r^j_{j'}, \cdot$) is specified.

The semantics of invocation on a scoped reference object are defined as follows. The invocation of a method on a scoped reference object throws `ScopedAccessException` if:

1. the referent has been cleared, or
2. the invoking thread is τ_i with current execution context $r^i_{i'}$, the referent is in region $r^j_{j'}$ where $i \neq j$ and *deallocateAfter*($r^j_{j'}, r^i_{i'}$) is not specified.

Otherwise, the invocation on the scoped reference object succeeds. A scoped reference object is cleared when:

1. the scoped reference object is in region $r^i_{i'}$, the referent is in region $r^i_{i''}$ and the deallocation point for $r^i_{i''}$ is reached, or
2. the scoped reference object is in region $r^i_{i'}$, the referent is in region $r^j_{j'}$, and for every *deallocateAfter*($r^j_{j'}, r^i_{r_x}$) specified, $r^i_{r_x}$ has finished executing.

6.1.3 An Evaluation of Scoped Reference Objects

The separation between the lifetime of regions and that of the objects they hold is achieved by scoped reference objects while providing failure semantics for invocations that are made on deallocated referents. The encapsulation of these references through a proxy means that a checked exception must be tested for in the event that the invocation fails. A closer investigation reveals that the semantics described above are consistent with the lifetimes of regions specified in the STS of a program in $\mathrm{RPM} - \mathrm{Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$. For scoped reference objects with referents in a different thread, failure of an invocation occurs when a region share is not specified between the current allocation context of the thread making the invocation and the region holding the referent. Therefore, the runtime lifetime semantics of regions differ from those specified in the STS of the program. Similarly, for scoped reference objects with referents in the same thread, failure of an invocation occurs when the runtime lifetime semantics of the region holding the referent differ from those specified in the STS of the program.[4] An important observation can be made that differentiates reference rules in the RTSJ from

[4]In this case however, the invocation might still succeed, even if the lifetime specified in the STS is violated. The occurrence of this phenomenon depends on the lifetime assigned to regions by the scoped ordering algorithm.

those in $\mathrm{RPM} - \mathrm{Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$ that use scoped reference objects. *In the RTSJ, the scoped memory references rules are used primarily to constrain the lifetime of objects; in* $\mathrm{RPM} - \mathrm{Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$ *programs that make use of scoped reference objects, the reference rules are also used as a runtime check that signals an error when the lifetimes specified in the STS differ from those observed during execution.*

The main disadvantage of the solution described here is that encapsulating references in a proxy means that fields of the referent that would be accessible if a strong reference were maintained are no longer accessible. Therefore it is required that in general, all fields are assumed to be only accessible by *get()* methods (that is they are all declared to be `private`). Since such an invocation on a scoped reference object would return a strong reference to any objects, the result would have to be assigned to a local variable before being assigned to the field of an object, potentially through the creation of another scoped reference object.

In developing applications that target $\mathrm{RPM} - \mathrm{Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$, the creation of scoped reference objects can be dynamic and automatic, with scoped reference objects being created as necessary when a strong reference would break the rules described above. The logic that is to be triggered when a `ScopedAccessException` is thrown can be specified at the required granularity (for example at a per-thread or per-region granularity). This approach eliminates the burden of identifying the objects that need to be referenced through scoped reference objects. This is important as the motivation for targeting a coarse grain memory model is to remove the burden of specifying any information on memory usage at a fine grain level. However, the use of scoped references objects to allow objects to reference each other adds a space overhead equal to the size of the proxy object and a time overhead on invocation equal to the cost of executing the logic that carries out the invocation. In order to derive the space and timing requirements of an application, it is therefore necessary to know which objects are to be referenced by scoped reference objects. A pessimistic assumption that all references are via scoped reference objects would mean that the memory required for a region is increased by a factor of the sum of the number of fields of all objects created in that region. Every invocation also incurs the additional execution cost of first invoking the proxy object. Although part of this overhead can be removed by using static analysis to identify references that are guaranteed to only need strong references, this problem is undecidable in general. The space and time overheads of using scoped reference objects to provide deterministic failure semantics bear a similarity to the overheads of tracing in automatic memory models as discussed in Section 2.1.3. In the former case, the overheads are incurred due to the requirement that invocations that follow dangling pointers fail deterministically; in the latter case the overheads are incurred due to the requirement that dangling pointers never occur. In both cases however, the additional overheads in comparison to a fine grain

Stricter semantics could be achieved by requiring that a scoped reference object be cleared when the deallocation point specified in its EDP specification is reached, rather than that derived in the RDP specification.

explicit memory are necessitated by safety requirements.

Finally, it is noted that the referents of strong references to scoped reference objects must be legal as per the RTSJ reference rules. Therefore, it is often the case, particularly for scoped reference objects with referents in the same thread, that the scoped reference object must be created in the allocation context of another region. This is the reason why the method *getProxy()* also takes an instance of `MemoryArea` as a parameter. In practice, this is achieved by using the RTSJ's *newInstance()* or *executeInArea()* methods of `MemoryArea`. This observation is relevant primarily to implementations of the RTSJ that provide scoped reference objects as classes directly available to developers. An implementation of reference objects for programs without region sharing was developed and is described in [35]. Of particular note is that this implementation was used to show how scoped reference objects can be used to implement thread groups. This is a particularly important example as thread groups are not supported in the reference implementation for the RTSJ [1]. This is because such an implementation in the presence of scope-using realtime threads would require the creation of references that would break the RTSJ rules.

6.2 Implementing $\mathrm{RPM-Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$ in a Java Environment

An implementation of a program that targets $\mathrm{RPM-Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$ in Java requires a way of describing these programs and expressing the lifetime of regions, a runtime that executes the program and makes the necessary calls to the underlying memory subsystem and an implementation of a memory subsystem. This section investigates how the first two of these items are achieved, the implementation of suitable memory subsystems having already been described in Sections 4.4 and 5.8. First, the classes required to express programs in $\mathrm{RPM-Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$ are introduced. A prototype implementation of a runtime that executes these programs is then developed. This runtime makes calls to the memory subsystem as required to allocate and deallocate memory for regions.

6.2.1 Describing $\mathrm{RPM-Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$ Programs and the Allocation/Deallocation Points and Synchronisation Points of Regions

Consider first the memory model $\mathrm{RPM-Scoped}^{\mathrm{cRPM}^{\rightarrow}}$ that describes the non multi-threaded version of scoped programs with a sequential flow. An abstract class `Region` is introduced that is extended by the class `RegularRegion`. An instance of `Region` references an instance of `Runnable` that describes the logic that is to be executed by that region. Every region r_i contains a reference to another region whose execution completion identifies the earliest deallocation point of r_i (that is, $\mathrm{EDP}(r_i)$). Since $\mathrm{RPM-Scoped}^{\mathrm{cRPM}^{\rightarrow}}$ programs specify a RAP and RDP as per the $\mathrm{cRPM}^{\rightarrow}$ LSL, two other references to objects are required: one that identifies where the **ENTER** annotation of r_i is (and hence its RAP) and the other that identifies where the **EXIT** annotation of r_i

is (and hence its RDP).[5] For RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ programs, region sharing and synchronisation must also be specified. Therefore, every region r_i also maintains a list of regions it synchronises with as specified by *sychroniseWith(r_i, r_j)* and any regions it must delay deallocation for as specified by *deallocateAfter(r_i, r_j)*. The logic to execute on a `ScopedAccessException` is specified by a `Runnable` instance at the granularity of a region.

The classes to describe program flow in RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ are based on the BNF notation introduced in Table 4.2 on page 95. A class `BlockList` is introduced for which instances maintain a reference to an instance of the abstract class `BlockType` and a reference to an instance of `BlockList`. There are three implementation classes of `BlockType`: `SeqBlock`, `LoopBlock` and `CondBlock`. Since only RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ programs are considered in this case, the `LoopBlock` and `CondBlock` are empty and would need to be implemented in future work to describe RPM – Scoped$^{\text{cRPM}^{\circlearrowleft}}$, RPM – Scoped$^{\text{cRPM}^{?}}$ and RPM – Scoped$^{\text{eRPM}}$programs. Thread instances of RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ are described by the class `RPMThread`, each of which maintains a reference to a `BlockList` instance. Finally, a program in RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ is specified by an instance of the class `RPMProgram` which is made up of a set of `RPMThread` instances. A UML class diagram of the classes described here is depicted in Figure 6.4.

An outline of the development process for applications that target RPM – Scoped LSLs is depicted in Figure 6.5. A tool that drives this development process and generates the instance of these classes automatically from some representation of a program could be developed. Although the development of such a tool is outside the scope of this thesis, Figure 6.5 highlights the functionality this tool might provide. First, a set of `Runnable` instances are developed that encapsulate the logic for the regions. Regions are then connected together to create the logic for a thread. Inter-thread synchronisation can also be specified at this point. The deallocation point of regions (their EDP) are then assigned and the correctness of these points are checked. Once all regions have correctly specified EDPs, the runtime allocation and deallocation points can be derived using algorithms such as *annotateWithEIA()* described in Chapter 5. Since RPM – Scoped is an explicit memory model, the logic to execute in the event of thrown `ScopedAccessExceptions` can be specified for each region. If this process is carried out without the use of a tool but directly through program logic, the result is similar to the code depicted in Figure 6.5. A tool would generate this code automatically from the visual representation of the program. A demonstration of this development process for an

[5] Note that in the prototype developed, every region maintains an array of **ENTER** and **EXIT** annotations rather than just one instance of each class as well as an array of regions that identify the EDP of that region. The size of these arrays is 1 in the case of RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$. This approach was adopted in preparation of the extension of the prototype to handle other LSLs of RPM – Scoped. It is also noted that in the special case of cRPM$^{\rightarrow}$ programs, all annotations result in entering or exiting a scoped region. This is not the case for other LSLs such as cRPM$^{?}$. Therefore every annotation instance in the prototype has a placeholder for arbitrary complex logic that is executed when an annotation is reached. This logic is described in the method *isValidInContext()* that returns `true` if the annotation is valid at the current point in the program's execution and `false` if it is not.

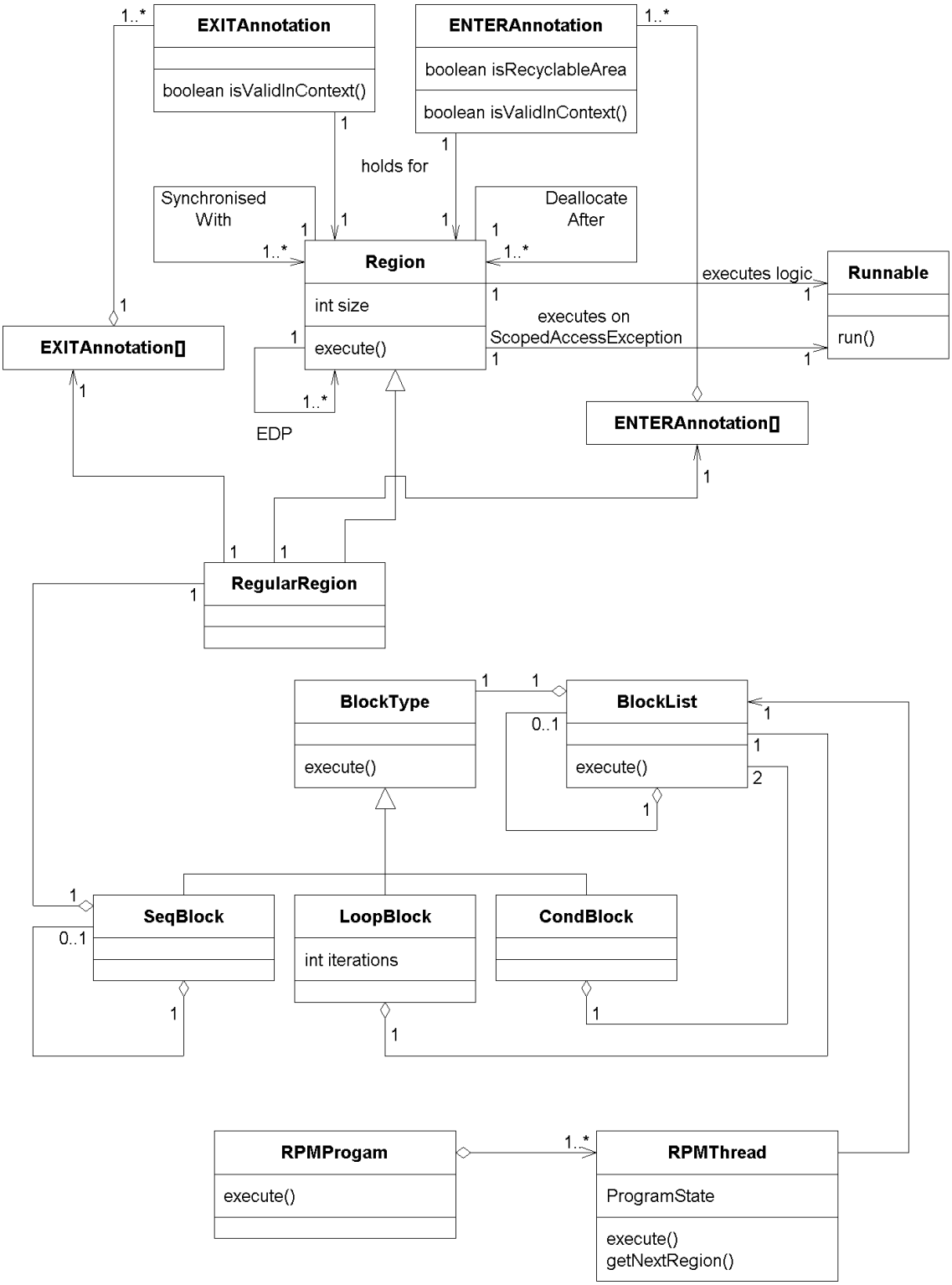


Figure 6.4: Class Diagram for Describing Programs in RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ (with Classes for Extension to Other Control Flows)

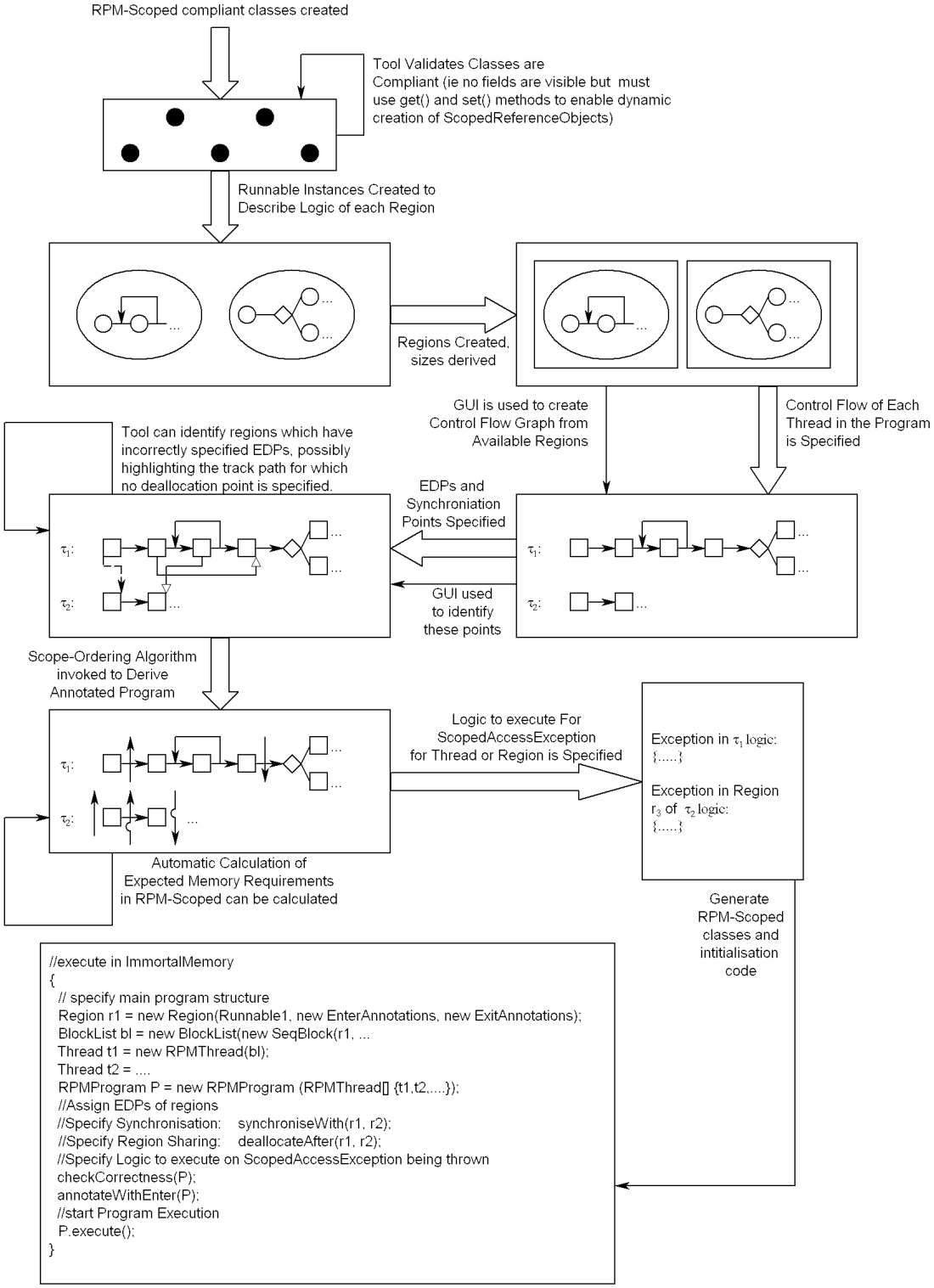


Figure 6.5: Developing Applications that Target RPM – Scoped

application that targets RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ is given in Section 7.1.

6.2.2 Executing RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ Programs in an RTSJ Environment

The execution of RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ programs can be implemented by middleware logic similar to EMM − ONLINE as described for RPM − Unscoped in Section 4.4. The control flow graph would be followed and the allocation and deallocation calls made to the memory subsystem based on the annotations at the inter-region locations. An alternative approach is to implement this runtime using the RTSJ where possible and extending it where necessary. The two extensions introduced in this thesis, scoped reference objects and recyclable memory areas clearly require extensions to the RTSJ. However, the logic for *enter()* and *executeInArea()* expressed in RPM − Scoped through **ENTER/EXIT** and **ENTER-EIA/EXIT-EIA** annotation pairs is already present in the RTSJ. Given an instance of `RPMProgram`, the initial memory map is set up by the memory subsystem as described in Section 5.8. The execution of the program and the calls to the memory subsystem are then handled using standard RTSJ *enter()* and *executeInArea()* methods. The only exception is when *executeInAreaClear()* is invoked on the extended `MemoryArea` class. A prototype of the logic required to execute an instance of `RPMProgram` that describes a program in RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ within an RTSJ environment has been developed and is described next.

Given an instance of `RPMProgram` allocated in immortal memory, invoking *execute()* on this instance in turns calls *execute()* of all instances of `RPMThread`. From a real-time perspective, this could be viewed as the change from the pre-mission to the mission phase. The execution of the `BlockList` instance associated with each thread is carried out by using a technique that separates the logic that identifies the next region to be executed from that which makes *enter()* and *executeInArea()* calls to the memory subsystem. The field `ProgramState` of `RPMThread` identifies the current state of the program that is relevant to the control flow of the thread. In the case of RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ threads, this is simply a reference to the current executing region.[6] The method *getNextRegion()* uses this state to identify the next region to execute. Again, this is trivial in the case of RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ threads, as it simply returns a reference to the next region in the `BlockList` instance.[7] Synchronisation is also handled by *getNextRegion()* by allowing this method to return a reference to a region $r_{i'}^{i}$ only when all $r_{j'}^{j}$ have finished executing for which $synchroniseWith(r_i^{i'}, r_j^{j'})$ is specified.

The execution of regions and invocation of *enter()*, *executeInArea()* and *executeInAreaClear()* is

[6] `ProgramState` is defined in order to allow for extensions to the prototype to consider other control flows. For example in RPM − Scoped$^{\text{cRPM}^{\circlearrowleft}}$ programs, `ProgramState` would also describe the current loop nesting of the current executing region whereas in RPM − Scoped$^{\text{cRPM}^{?}}$ it would include the result of the execution of past conditional regions.

[7] *getNextRegion()* is also defined in order to allow easy extension of the prototype to other control flows. For example, if the current executing region is the last region in the `BlockList` instance of a `LoopBlock`, then the returned region is the first region in this `BlockList` if the program has not yet executed all iterations of the loop.

handled by a class called `Runner`. *executeInAreaClear()* is not considered here in order to simplify the description of this class. The non-nested nature of the way annotations are read after invoking *getNextRegion()* must be converted to a nested structure in order to make use of these methods. This is achieved by making the invocation stack of the thread equivalent to the nesting of scoped memory regions. The class `Runner` implements the `Runnable` interface and the logic of the *run()* method guarantees this nesting. Code listing 6.1 shows the implementation of `Runner` for executing threads of RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ programs. The first instance of this class is created by `RPMThread`.*execute()* with the first parameter being a reference to the first region of the program. Between Lines (15) and (21), the *run()* method of this instance is invoked recursively for each **ENTER** annotation found at the inter-region location before the region associated with this instance of `Runner`. The allocation context at each invocation is changed to that of a new scoped memory instance created at Line (19).[8] Each region is associated with its instance of `ScopedMemory` at this stage.[9] Once all **ENTER** annotations have been processed, the region associated with the current instance of `Runner` (`this.r` in Line (23)) can be executed in the context of its assigned memory area (Line (24)). Once this execution terminates, the **EXIT** annotations found at the inter-region location after the region associated with this instance of `Runner` must be processed. For each one, invoking *return* at Line (28) pops the current invocation stack and a scoped memory area since an invocation of *enter()* at Line (21) returns. Once all **EXIT** annotations have been processed, the next region to be executed is obtained in Line (31) and the process is repeated. If *getNextRegion()* returns `null` at Line (31), then the program has finished executing and can exit. Implementations of `Runner` for RPM − Scoped$^{\text{cRPM}^{?}}$ and RPM − Scoped$^{\text{cRPM}^{\bigcirc}}$ are similar. The processing of an **ENTER** and **EXIT** annotations must however be preceded by a call to *isValidInContext()* and, if this returns `false`, then that instance of the annotation is ignored.

6.3 Comparing the RTSJ Memory Model to RPM − Scoped

The RPM − Scoped memory model was developed with the overriding goal of reducing space and time overheads due to fragmentation by adopting a scoped ordering. When such an ordering can not be described deterministically due to the dependence of the lifetime of a region on the progress of other threads, this must be stated explicitly. For example, this is achieved in RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ by using the method *deallocateAfter()*. The memory subsystem developed in Section 5.8 is designed to make use of this information in order to reduce fragmentation. A secondary goal of the RPM − Scoped memory model was to take advantage of expressed synchronisation in order to reduce

[8] Note that, although not shown here, it is necessary to check here whether the **ENTER** annotation has the `isRecyclable` field set, in which case *executeInAreaClear()* is called in Line (24) rather than *executeInArea()*.

[9] It is noted in passing that a Java-level implementation requires scoped reference objects in this case since the instances of `Region` are in immortal memory whereas the instances of `ScopedMemory` are themselves in scoped memory areas.

Listing 6.1: Implementation of `Runnable` for RPM – Scoped cRPM$^{\rightarrow}$

```
1
public class Runner implements Runnable {
3     public RegularRegion r;
      public int ENTERcount;
5     public RPMThread rpmthread;

7
      public Runner(RegularRegion r, RPMThread t) {
9         this.r = r;
          this.ENTERcount = 0;
11        this.rpmthread = t;
      }
13
      public void run() {
15        if (this.ENTERcount < r.enterannotations.length) {
              Region RegionToEnter = r.enterannotations[ENTERcount].
17                                   holdsForRegion;
              ScopedMemory SM = new LTMemory(RegionToEnter.size, this);
19            RegionToEnter.myMemoryArea = SM;
              this.ENTERcount++;
21            SM.enter();
          } else {
23            ScopedMemory SM = this.r.myMemoryArea;
              SM.executeInArea(this.r.getRunnable());
25        }
          if (this.rpmthread.pending_returns > 0) {
27            this.rpmthread.pending_returns--;
              return;
29        }

31        RegularRegion nextregion = this.rpmthread.getNextRegion();
          if (nextregion != null) {
33            Runner r = new Runner(nextregion, this.rpmthread);
              this.rpmthread.pending_returns = nextregion.exitannotations.
35                                             length;
              r.run();
37        }
      }
39 }
```

memory requirements. Again, the benefits of expressing this information are realised in the memory subsystem. The differences between RPM − Scoped and the scoped memory model presented in the RTSJ can be observed at two levels: firstly, in the expression of the lifetime of regions and secondly in the way implementations are required to make use of this information. Scoped reference objects and recyclable memory areas are two examples of the former in that they are extensions to the RTSJ in order to describe RPM − Scoped programs. The key differences between the two models at these levels are investigated in further detail next.

A significant difference between these models is that a region in the RTSJ can be shared by two or more threads, leading to a cactus-stack structure of scoped memory areas. As described in Section 2.2, two threads can therefore share the same allocation context, thereby both creating objects in the same physical memory area. The motivation for this approach is to allow threads to share data, as strong references are legal in the same memory area. RPM − Scoped achieves the same functionality using scoped reference objects and explicit specification of region sharing. However, whereas the RTSJ requires that all outer regions of the location of the shared data also be shared between the two threads, this is not so in RPM − Scoped. Therefore, whereas the lifetime of only that region where the shared object resides can not be derived exactly in RPM − Scoped, in the RTSJ memory model, the lifetime of all inner regions also can not be determined exactly. This therefore results in a greater degree of fragmentation in the memory subsystem.

Another difference between the two models is that RPM − Scoped does not distinguish between the lifetime of a scoped memory object and its backing store. Indeed, in the prototype implementation described in Section 6.2.2, an instance of `LTMemory` is created at the point when the region needs to be entered. However, since RTSJ semantics dictate that exiting a region does not deallocate its backing store but only clears it, a modification is required in these semantics so that exiting a region also results in deallocation of the backing store. The distinction made between clearing a region and deleting its backing store is not required in RPM − Scoped as a memory area can only ever be the allocation context for a single thread. The semantics defined by the RTSJ result in a separation between the lifetime of a region and that of the memory it executes in. This makes it harder to derive the worst case memory requirements of an application and to develop an allocation and deallocation strategy that minimises fragmentation.

The RTSJ does not specify in any way how its memory subsystem is to be implemented. Indeed, most implementations invoke the operating systems *malloc()* and *free()* systems calls whenever the semantics of the model dictate that these are required. The intra-thread fragmentation due to these semantics as described above are therefore further exacerbated as the memory for different threads is not partitioned as described for RPM − Scoped's memory subsystem in Section 5.8. This results in the memory subsystem performing in the worst case as badly as that of EMM − ONLINE. As

argued in Section 4.4, EMM − ONLINE can perform as badly as a fine grain memory model in the space requirements of the application. In such implementations, the RTSJ therefore fails to take advantage of the key advantage of the scoped approach: the ordered allocation and deallocation of regions. The result is a model which introduces additional complexity in requiring the aggregation of region lifetimes and their ordering to be specified but then fails to take advantage of any of this information at runtime.

6.4 Summary

This chapter investigated the implementation of the RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ memory model in an RTSJ environment. First, an extension to the RTSJ scoped reference rules was introduced in order to allow the lifetime of regions to be expressed independently of the object graph. Classes were introduced that allow the STS of programs in RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ to be described in Java and the general development process for applications that target RPM − Scoped programs was described. A prototype implementation of a runtime that takes an STS of an RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ program described through these classes and executes the program using existing RTSJ routines whenever possible was described. In the next chapter, a case study of an application developed to target RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ is carried out that follows the development process described here. Finally, this chapter investigated the key differences between RPM − Scoped and the RTSJ scoped memory model. It was argued that the RTSJ's approach whereby regions that share objects must also share memory areas leads to indeterminacy in the lifetime of regions, making it impossible for the memory subsystem to adopt an allocation and deallocation strategy that eliminates fragmentation. It was also argued that the failure of existing implementations of the RTSJ to take advantage of the scoped ordering of regions results in runtime space requirements that fail to improve on those incurred in fine grain models.

Chapter 7

Evaluating the Scoped, Coarse Grain Memory Model for Real-Time Java Applications

Section 6.2 of the previous chapter showed how real-time Java applications that target RPM – Scoped are developed. This chapter begins in Section 7.1 with a demonstration of this approach by developing a simple application that targets RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ in a Java environment. The separation of program logic from the expression of region lifetime as well as the separation of region lifetime from the object graph is shown to be important in simplifying development and maintenance. Recall that the runtime for RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ is in fact built on top of an extended RTSJ. It is therefore concluded that, at least for multi-threaded programs with terminating threads[1] that execute regions in sequential order, the complexity of developing and maintaining RTSJ programs is an inherent symptom of this development approach and the unnecessary restrictions of the RTSJ scoped reference rules, rather that a symptom of the scoped, coarse grain approach. In Section 7.2, the analysis on the scoped approach to a coarse grain model carried out in Section 5.6 forms the basis of a criticism of the RTSJ's Expert Group decision to adopt a scoped, coarse grain model rather than an unscoped one. In particular, it is important to reflect on the conclusions that might be drawn from this investigation in relation to the two questions posed at the beginning of this thesis (see page 2). Finally, Section 7.3 revisits the thesis hypothesis in order to show how the contributions provided in this thesis prove each of the three parts of this hypothesis.

[1] The additional complexity caused by a scoped order in non-terminating threads, and that requires the developer to map this model to other models [24] is addressed by recyclable memory areas.

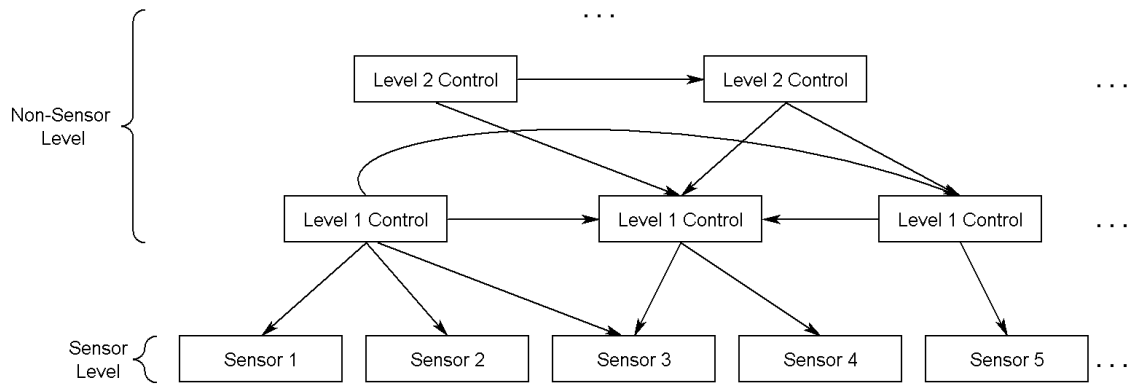


Figure 7.1: Abstract Model for a Hierarchical Control System

7.1 Developing A Hierarchical Control System Targeting RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$

In Section 6.5, a general development process for applications that target RPM − Scoped was outlined. The case study developed in this section demonstrates how an application that targets RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ would be developed using this process. The importance of separating memory concerns from program logic as well as separating region lifetime from the object graph is discussed.

Consider an abstract hierarchical control system [10] as shown in Figure 7.1. Each task in the sensor level receives data from a sensor and carries out some preliminary processing of this data. Tasks in the lowest level of the non-sensor level, make use of the final result of data produced in a subset of the sensor tasks as well as results produced by tasks at the same level. This result can be considered to be some representation of the current state of that component. Tasks at higher levels make use of the result of tasks at the same and lower levels in the system.[2] Based on the result of any task, a set of actuators may be triggered. All tasks can be assumed to be periodic, creating a final result that is used by other tasks in the system and possibly triggering actuators. Large-scale control systems typically possess such a hierarchical architecture in order to manage complexity and many examples of such systems are found in the literature (for example [124]). An example of a realisation of the abstract hierarchical control system is shown in Figure 7.2. In this example, a set of sensors monitor the state of an aircraft. Higher-level control routines use this data in order to make any required changes to the state of the system. For instance, engine power may be changed based on the recorded pressure, weight and height of the aircraft. The state of engine power is encapsulated in an object that is used, together with data from the sensors of weight, height

[2]Traditional hierarchical control systems as described in [10] are strictly hierarchical in that a task communicates only with tasks in the level immediately preceding it. The model described here is an extension to these semantics that provides additional flexibility.

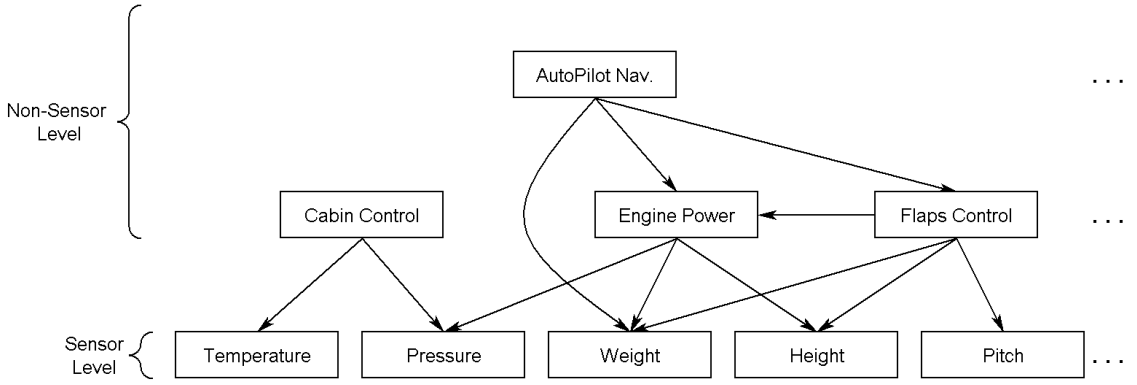


Figure 7.2: Part of An Aircraft System Modelled as a Hierarchical Control System

and pitch, to derive a suitable state for the flaps. Actuators then move these flaps and their state is saved. Similarly, the autopilot navigation system uses the state of the engine power and flaps together with the weight of the craft to plot a course to the destination.

As discussed in Section 3.2, the decision to target a memory model must be made before implementation of the system begins. In this case, the decision to target a coarse grain memory model will be based on whether the application exhibits object lifetime aggregation across program flow. In the hierarchical control system being developed here, a design decision is taken whereby every task is divided into three "processes". At every release, the task first reads state data from sensors and other tasks in the system, creating the state on which it is to process its own result. Next, the main processing of this data begins with the task generating temporary objects as necessary. Finally, the last process involves the generation or updating of an object that encapsulates the state of that task. Therefore, since the lifetime of objects in each process does exhibit lifetime aggregation against program flow, the RPM (and indeed the RTSJ) is a suitable model to target.

Testing the Implementation in a GC Environment

The implementation of the system begins with the implementation of each process in a similar way to that in a garbage collected environment with all standard Java classes being available for use.[3] As all tasks in the Hierarchical Control System are made up of the same three processes, an interface `ControlTask` is defined that describes each of these processes. All control tasks implement this interface in order to describe the logic that is executed in each of the three processes. An initialisation method must also be implemented to set up any data structures for sharing immortal objects. This method is executed in the pre-mission phase and the allocation context for this process is therefore assumed to be immortal memory. Finally, a method *GetResult()* is required that returns a reference

[3]This includes utility classes in the `java.util` package such as the `Vector` class. These classes in particular must be used with caution in the RTSJ but not in RPM – Scoped.

to an object that represents the state of the control task. As *GetResult()* will shortly be explained in more detail it suffices to note here that the object returned from this method encapsulates the state of the task that is shared with other tasks in the system. The interface `ControlTask` is defined as follows:

```
public interface ControlTask {
    public void Init();
    public void ReadData();
    public void ProcessData();
    public void GenerateResult();
    public Object GetResult();
}
```

Before describing this program using RPM notation, it would be useful to test the functionality of the application in a garbage collected environment. This can be done by describing the control flow of the application using standard Java classes. Since all tasks follow the same flow, a single class can be defined that wraps each of these methods in a `Runnable` instance. This is achieved by the abstract class `TaskRunner` that is itself a `Runnable` instance as it defines the logic for an RPM thread. This logic is to first call *Init()* and then enter an infinite loop in which *ReadData()*, *ProcessData()* and *GenerateResult()* are called in turn:

```
public  class TaskRunner implements Runnable{

    ControlTask controltask;

    public TaskRunner(ControlTask t)
    {
        this.controltask=t;
    }

    public void run()
    {
        this.controltask.Init();
        while (true)
        {
            this.controltask.ReadData();
            this.controltask.ProcessData();
            this.controltask.GenerateResult();
        }
    }
}
```

At this point, the functionality of the system can be tested by creating an instance of `TaskRunner` for each task, encapsulating this instance in a `Thread` instance and then starting this thread.

Specifying the Control Flow of Tasks in RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$

The next stage is to convert the program into an equivalent RPM – Scoped program using the classes defined in Section 6.2. This allows the specification of region lifetimes and the derivation of RAPs and RDPs (through `ENTERAnnotation` and `EXITAnnotation` instances) to drive allocation and deallocation without assuming a garbage collector. In this case, the version of RPM – Scoped required is one which allows the specification of loops within each thread. However, since only classes for targeting RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ and a runtime for these programs have been developed, this example must be restricted to this model. For the purposes of this demonstration, this is not a serious problem since the benefits of the development process for targeting RPM – Scoped are still evident. The hierarchical control system is therefore restricted from one describing infinite tasks to one in which tasks execute for a fixed number of iterations. The `while (true);` statement of the class `TaskRunner` is therefore changed to a `for` loop with a fixed number of iterations. The description of the control flow of this program in RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ in tantamount to the unrolling of this loop for the specified number of iterations.

In order to target RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$, each task that until this point is described by `TaskRunner` must be transformed into an instance of `RPMThread`. Therefore, for every control task, a new `RPMThread` instance is created to describe the control flow of that task. First, the definition of region boundaries by `Runnable` instances is required for every one of the first four methods described in the `ControlTask` interface. The control flow of the task is then defined by the instance of `BlockList` encapsulated in the instance of `RPMThread`. As an example, the pressure sensor task in the Aircraft System in Figure 7.2 is represented in RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ in the following listing. Note that since RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ only allows the `BlockList` instance of the `RPMThread` to hold `SeqBlock` instances, the regions of the program are created in an array called `regions[]`. This captures the loop unrolling described above.

```
final PressureSensor S = new PressureSensor();

 Runnable ReadDataRunnable = new Runnable() {
 public void run() {S.ReadData();} };

 Runnable ProcessDataRunnable = new Runnable() {
 public void run() {S.ProcessData(); } };

 Runnable GenerateResultRunnable = new Runnable() {
 public void run() {S.GenerateResult(); } };

 int number_of_iterations = 5;
 Region[] regions = new Region[number_of_iterations * 3];
 SeqBlock seqblock = null;

 for (int i = number_of_iterations - 1; i >= 0; i--) {
```

```
    Region ReadDataRegion = new RegularRegion(ReadDataRunnable);
    Region ProcessDataRegion = new RegularRegion(ProcessDataRunnable);
    Region GenerateResultRegion = new RegularRegion(GenerateResultRunnable);
    regions[(i * 3)] = ReadDataRegion;
    regions[(i * 3) + 1] = ProcessDataRegion;
    regions[(i * 3) + 2] = GenerateResultRegion;
    seqblock = new SeqBlock((RegularRegion) ReadDataRegion,
          new SeqBlock((RegularRegion) ProcessDataRegion,
          new SeqBlock((RegularRegion) GenerateResultRegion, seqblock)));
    }
    BlockList bl = new BlockList(seqblock);
    RPMThread sensorthread = new RPMThread(bl);
```

Specifying Region Lifetime to target $\mathrm{RPM-Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$

The next stage in the development process is to specify the relative lifetime of regions. This information will be based on the developer's knowledge of the underlying operation of the task; that is the lifetime of regions is already implicit in the application – the developer only needs to express this information. First, the specification of the lifetime of a region relative to other regions in the same task is required. As an example, consider that the pressure sensor task in the Aircraft System operates as follows. In the *ReadData()* phase, three pressure sensor gauges are sampled and an array object that encapsulates this data is created. In the *ProcessData()* phase, a new object is created that merges data generated in the previous *ProcessData()* invocation with the latest raw data. Finally, the necessary actuators are activated and a new object is created in the *GenerateResult()* phase that represents the pressure state of the aircraft that is to be shared with the other tasks in the system. The code listing for the pressure sensor task is therefore as follows:

```
public class PressureSensor implements ControlTask {
    public int[] rawdata;
    public Integer latestread=new Integer(0);
    public Integer sharedstate;

    public PressureSensor()  { }
    public void Init() {}
    public void ReadData() {
        rawdata = new int[3];
        rawdata[0] = PressureSensorGuage1.sample();
        rawdata[1] = PressureSensorGuage2.sample();
        rawdata[2] = PressureSensorGuage3.sample();
    }
    public void ProcessData() {
        latestread = new Integer((this.latestread.intValue() + this.rawdata[0] +
                                  this.rawdata[1] + this.rawdata[2]) / 4);
    }
    public void GenerateResult() {
        this.sharedstate = new Integer(this.latestread.intValue());
    }
    public Object GetResult(int x) {
```

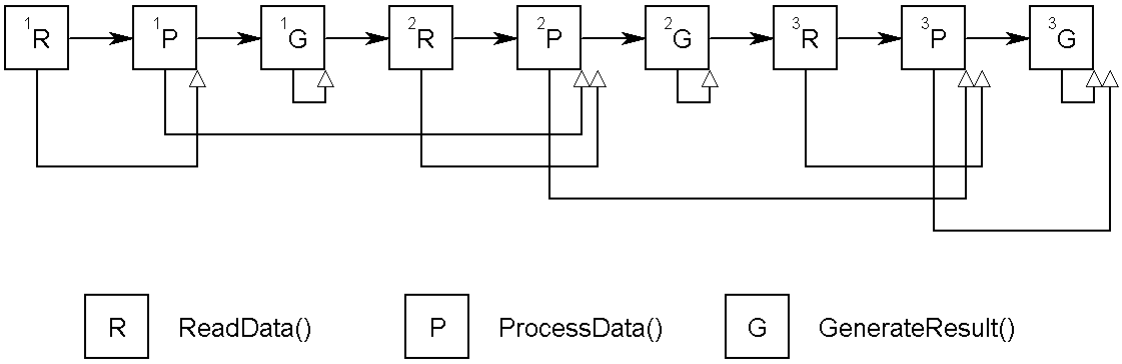


Figure 7.3: The Lifetime of Regions in The Pressure Sensor Task

```
        return this.sharedstate;
    }
}
```

The lifetime of each region is specified by setting the deallocation point of that region. In this case, the EDPs of regions for the program are as depicted in Figure 7.3 where three iterations of the loop are unrolled. Since the developed prototype only allows the specification of RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ programs, this point must be specified for each region in turn. In an extended implementation that recognises RPM − Scoped$^{\text{cRPM}^{\circlearrowleft}}$, a representation of EDPs in the cRPM$^{\circlearrowleft}$ LSL would be provided instead. The code listing for the assignment of these deallocation points follows:

```
for (i = 0; i < number_of_iterations-1; i++) {
      ((RegularRegion) regions[i*3]).EDP= ((RegularRegion) regions[(i*3)+1]);
      ((RegularRegion) regions[(i*3)+1]).EDP= ((RegularRegion) regions[((i+1)*3)+1]);
      ((RegularRegion) regions[(i*3)+2]).EDP= ((RegularRegion) regions[(i*3)+2]);
}
((RegularRegion) regions[i*3]).EDP= ((RegularRegion) regions[(i*3)+1]);
((RegularRegion) regions[(i*3)+1]).EDP= ((RegularRegion) regions[(i*3)+2]);
((RegularRegion) regions[(i*3)+2]).EDP= ((RegularRegion) regions[(i*3)+2]);
((RegularRegion) InitRegion).EDP= ((RegularRegion) regions[(i*3)+2]);
}
```

Having specified the lifetime of regions relative to other regions in the same thread, the next stage is to specify cross-thread region sharing depending on how data is shared between tasks. In this example, this involves describing what *GetResult()* returns on invocation. There are at least two approaches that could have been adopted, both of which can be described in RPM − Scoped$^{\text{cRPM}^{\|\sigma\rho}}$. The choice of either approach may be dictated by the specification of the application, rather than the developer's preference. These two approaches are investigated next.

The first approach to sharing the object representing the state of a task is to have a single instance of the shared object that is protected from concurrent access by means of a monitor. This is the simplest approach to sharing the state of each task and requires no cross-thread region sharing

or synchronisation to be specified. In the example of the pressure sensor task given above, the shared object would be represented by a wrapper of the `Integer` class that provides `synchronized` methods to read and write to instances of this class. *Init()* would create this shared object that is allocated in immortal memory. *ProcessData()* would update this value on a new result being calculated and *GenerateResult()* would return a handle to this shared object. The implementation of these methods would be as follows rather than that listed in the previous code listing:

```
public void Init()
{
    this.sharedstate = new SynchronizedInteger(0);
}

public void GenerateResult() {
    this.sharedstate.update(this.latestread.intValue());
}
public Object GetResult() {
    return this.sharedstate;
}
```

An alternative approach is to have each task sample the result derived at the same release iteration as that of other tasks it samples. For every task i that requires access to the shared state of another task j, region sharing and synchronisation between the two tasks must be specified. For example, assume that it is specified that if Task 1 uses the state of Task 2, then the state sampled by Task 1 at release i is that generated by Task 2 at its i^{th} release. In order to ensure Task 2 only need maintain one instance of its current state, it is also required that Task 2 does not begin its $i^{th}+1$ release of *GenerateResult()* before the i^{th} release of *ProcessData()* in Task 1 has finished executing. One advantage of this approach over sharing a single object through immortal memory is that blocking does not occur in the task creating the shared instance. However, the semantics of this approach dictate that a task must wait for the shared data it needs to be created and can therefore not just use the most recently available data.

Using the example of the pressure sensor task, the original code listed above requires no modification. What must be specified are the region sharing and synchronisation semantics of tasks that require the shared data generated by this task. This is achieved through the *synchroniseWith()* and *deallocateAfter()* methods. As an example, the control flow graph for two iterations of the pressure sensor task together with region sharing and synchronisation points for a task that uses the state of this task in depicted in Figure 7.4. From Figure 7.2 this task might be the "Cabin Control" or "Engine Power" tasks. The functionality of these tasks is irrelevant in this example; it must only be known that they sample the state generated by the pressure sensor task using the semantics described above. Since the developed prototype only allows the specification of $\mathrm{RPM}-\mathrm{Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$ programs, region sharing and synchronisation points must also be specified for each region in turn

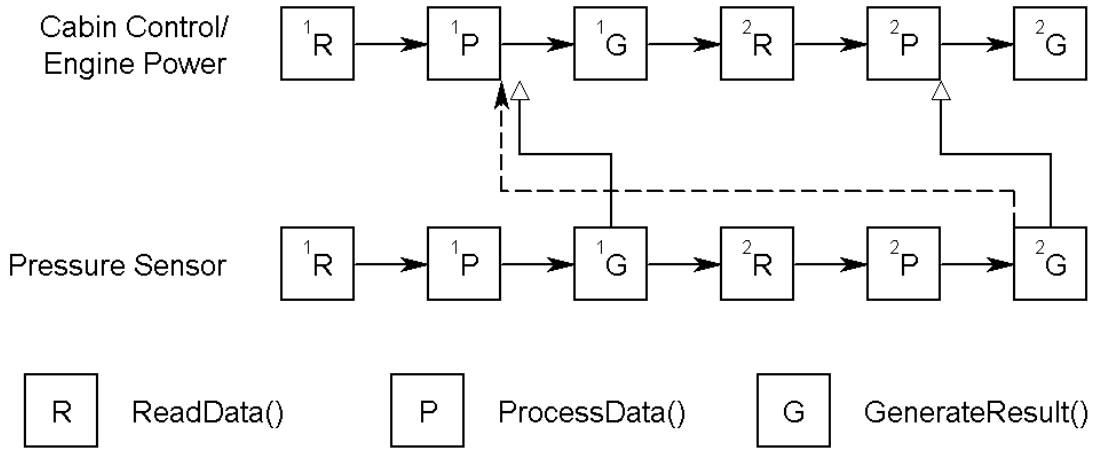


Figure 7.4: Expressing Region Sharing and Synchronisation with The Pressure Sensor Task

in a similar way to how EDPs were specified.

There are several other possibilities that could have been adopted to describe how the state of a task is shared and that can be described as a combination of region sharing and synchronisation. For example, if it were required that the latest version of a result object had to be returned on *GetResult()* being called, then a buffer of result objects could be kept. The size of this buffer would be guaranteed not to be overrun by specifying suitable synchronisation points that ensure the task can not progress into a new invocation of *GenerateResult()* until other tasks that require the state of this task have progressed sufficiently. The previous example is a special case of this approach with a buffer size equal to 1.

Deriving a Scoped Ordering and Launching the Application

At this point, all tasks have been specified together with the deallocation points of their regions (as specified by the EDP field for the deallocation point in the same thread and *deallocateAfter()* for that in other threads) and the necessary synchronisation points. The penultimate step is to consider whether any specific logic is to be executed in the event that a `ScopedAccessException` is thrown due to an incorrectly specified deallocation point. For this simple example, no such logic is specified. The final step is to encapsulate each of the control tasks in an instance of `RPMProgram`, check this instance for correctness and finally invoke *annotateWithEIA()* if the deallocation points are indeed specified correctly. The program is now a valid $\mathrm{RPM} - \mathrm{Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$ program which can be executed when the mission phase begins using the *execute()* method:

```
boolean missionphase=false;
RPMProgram P = new RPMProgram(new RPMThread[]{sensorthread,...});
if (P.checkForCorrectness())
{
```

```
    P.annotateWithEIA();
    sensorthread.Init(); //... followed by other thead initialisers
    while (!missionphase); //... wait for start of mission phase
    P.execute();
}
else System.out.println("Invalid Program");
```

Evaluating the Development Process for Targeting RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$

The example described in this section demonstrates how targeting RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ need not require the developer to consider the order of regions within the scope tree. Given that the application is designed to target a coarse grain model, the only additional requirement in comparison to a development process that assumes a garbage collector is the specification of region boundaries and the lifetime of these regions. Any changes to the application need only be captured by changing these two pieces of information. For example, if the pressure sensor task no longer uses data generated in the previous *ProcessData()* invocation, then changing the EDP of the associated region suffices. Similarly, if the semantics for sharing the result objects changes, this is captured by specifying new region sharing and synchronisation behaviour using *deallocateAfter()* and *synchroniseWith()* as described above.

The separation between the specification of memory usage in the application and the chosen memory subsystem is captured in the way RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ programs are specified. The decision to adopt a scoped model is only taken when `P.`*annotateWithEIA()* is invoked. Using the same region lifetime information specified until this point, alternative memory models could be used. For example, if a scoped ordering when using SMM$^{\|\sigma\rho}$ is known to incur higher memory requirements than using EMM – ONLINE,[4] then the latter memory subsystem could be used instead. This requires no changes in the application's logic or the specification of region boundaries and region lifetime.

A key benefit of RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ is that the restrictions posed by the RTSJ reference rules are eliminated through the use of scoped reference objects. It could be argued that these rules could be maintained in RPM – Scoped$^{\text{cRPM}^{\|\sigma\rho}}$ by updating specified region lifetime when the object reference graph is known to have changed. However, this would also result in region merging in the case when bi-directional references are created, an operation tantamount to changing region boundaries. This phenomenon occurs frequently in RTSJ development, resulting in manual reworking of regions. The development of the above application demonstrates the degree to which the elimination of this restriction simplifies application development. Note in particular that the implementation of the logic in each region was developed without considering whether a scoped or unscoped memory subsystem was to be used. Indeed, the same logic tested in a garbage collection

[4]For example, if all regions have the same size, then EMM – ONLINE may be decided to be more appropriate since fragmentation never occurs.

environment is used in targeting $\mathrm{RPM-Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$ and no restriction on the use of library classes is required. Targeting a coarse grain model does require an additional development burden in that region boundaries and their lifetimes must be specified. However, requiring the developer to identify these boundaries during the design process by considering implementation specific information (in particular the object graph) results in a complex development process that is perhaps the major detraction of the RTSJ.

Finally it is noted that recyclable memory areas are equally important to scoped reference objects in targeting a scoped model. Since the example developed here consists of terminating threads, recyclable memory areas are not necessary since memory requirements are finite. Their importance in targeting a scoped model has been adequately covered in Section 5.6. It suffices to reiterate here that recyclable memory areas allow the specification of patterns of region lifetime that would otherwise lead to infinite memory requirements in a strictly scoped model whilst still guaranteeing the absence of fragmentation.

7.2 A Criticism of the Scoped, Coarse Grain Memory Model and the RTSJ's Approach

Recall that in Chapter 1, it was noted that achieving a memory model that can provide predictability and low space and time overheads is an objective not unique to real-time Java developers but also to the wider real-time and embedded systems community. Two key questions were posed:

(Q1): What is the best memory model to use in a particular class of real-time systems?

(Q2): What is the best memory model to use given the syntax and semantics of the Java Language and Virtual Machine Specifications?

In developing the RTSJ, the Expert Group chose to propose a new memory model as the current state-of-the-art in garbage collection technology was considered unsuitable for the application domain they wished to target.[5] The detailed investigation carried out in Section 2.1 indicates that this conclusion is correct. The challenge would then have been to provide answers to both the above questions. Proposing a static memory model as an answer to the first question would have been the simplest option. This would mean that the RTSJ would follow a practice common to other development environments. Moreover, this approach would require no changes to Java's memory model, therefore making a trivial case for answering the second question.

The Entropy Hypothesis has been used throughout this thesis to make the case for allowing more information on memory usage by the application to be expressed. The RTSJ acknowledges

[5]It is reiterated that this application domain is understood to include hard real-time and embedded systems that are currently developed in Ada and C.

this by providing a memory model that allows the developer to capture object lifetime aggregation against program flows. However, as argued in Section 3.2, when this phenomenon is not implicit in the application, targeting this model provides no benefit but only increases the development burden. The result is a library of "anti-patterns" as reviewed in Section 2.3.1 that attempt to express incompatible memory usage information in a coarse grain model. One way of avoiding this problem is to develop the application to immediately target this model. This is an argument that is common amongst advocates of the RTSJ model. The example given in Section 7.1 demonstrated this approach by dividing each task into processes. However, it is not possible to assume this approach can be adopted for all systems. Memory management is only one concern for the developer and other concerns, such as performance requirements may need to take preference in the way the application is designed.

Although the RTSJ Expert Group's adoption of a coarse grain model has been argued in this thesis to be a positive step, the decision to adopt a scoped approach has often been criticised [5]. The rationale for this decision is that it provides a framework on which the scoped reference rules are defined and by which dangling pointers are guaranteed never to occur. However, as shown in Section 6.1, this creates unnecessary coupling between the specification of the lifetime of regions and the reference graph. As the solution proposed in Section 6.1.2 could also be adapted to a coarse grain model without a scoped order, it is necessary to investigate whether a memory subsystem that assumes such an ordering provides any benefit over one that uses a fixed allocation offline algorithm to determine where to allocate memory.

When targeting a scoped memory model using either an automatic scope ordering algorithm such as *annotateWithEIA()* or a manual approach, the developer is still required to know the exact lifetime of each region. This information is identical to that required when using an offline allocation policy. The key difference between a memory subsystem that uses an unscoped offline algorithm and one that uses a scoped offline algorithm is that the former derives a location in memory in which to allocate memory for each region whereas the latter specifies when memory for a region is to be allocated. Therefore, in a memory subsystem for a scoped model, when an allocation point is reached, the location where memory for that region is to be allocated is not pre-calculated but is identified by the top-of-stack pointer for each scope stack. There are two immediate advantages of this approach. First, the physical memory space used by each task can be shared in the memory subsystem as described in Section 5.8. This provides a simple way of taking advantage of expressed synchronisation that existing offline allocation algorithms are unable to achieve. The second advantage of a scoped approach is that additional flexibility can be obtained by allowing the size of a region to be specified as late as its RAP. For example, consider two regions in a thread that have identical allocation and deallocation points and variable sizes that however sum up to a fixed size. If this information could

be expressed in an STS, then a scoped model will automatically take advantage of this information. However, since existing offline analysis requires the size of each region to be fixed in order to derive an allocation strategy, then it will have to be assumed that the memory requirements of each region is equal to the maximum possible size. The main disadvantage of a scoped model in comparison to using a fixed allocation offline algorithm has been proven to be the possibility of increased, and sometimes infinite, memory requirements. This was argued to occur due to the inherent inability of the model to retain the exact lifetime of every region. Although recyclable memory areas alleviate this, the memory requirements of a memory subsystem that uses this model may still be greater than with a fixed allocation policy.

The answers to (Q1) and (Q2) above can be argued by the Entropy Hypothesis. Assuming an application does exhibit object lifetime aggregation against program flow, a coarse grain memory model as described by the RPM in Chapter 4 is a suitable memory model for hard real-time systems. It is however not suitable for those applications that do not exhibit this aggregation. Therefore a coarse grain memory is a *partial* solution to (Q1). The RTSJ attempts to provide a response to (Q2) by means of its scoped model that captures lifetime information and provides safety against dangling pointers with the same mechanism. This is the cause of the development complexity experience by RTSJ developers. In the example in Section 7.1, it was shown that separating these two mechanisms eliminates this complexity. However, it would be incorrect to therefore conclude that scoped ordering of regions should be eliminated. As noted above, there are advantages in adopting a scoped ordering that can not be leveraged by memory subsystems with on-line or offline policies. The solution for (Q2) in the particular case of applications that target a coarse grain models must therefore lie in providing a way of expressing all this information. The memory subsystem would then adopt a mixed approach that uses a scoped ordering based on the algorithms developed in this thesis, a fixed ordering based on traditional offline algorithms or a dynamic on-line policy. The specific policy chosen in the memory subsystem for each thread (or even a granularity finer than a thread) would then depend on what information is specified for the regions of that thread. As dictated by the Entropy Hypothesis, this approach would thereby leverage the maximum advantage in reducing space and time overheads that depends on the specified information.

7.3 Evaluating the Thesis Hypothesis

The thesis hypothesis defined in Chapter 1 was divided into three parts. This section highlights how the contributions in this thesis prove each of these three parts.

Part 1: As hard real-time systems become increasingly large and complex, forcing a static model of

memory usage becomes untenable. The challenge is to provide a memory model that guarantees reasonably tight and bounded time and memory requirements without overburdening the developer with memory concerns. This thesis contends that a coarse grain approach can help *meet this challenge. In particular, program development and maintenance is aided by providing an environment that allows the expression of the lifetime aggregation of objects and the interaction of these aggregates as a separate concern to program logic.*

The case for a coarse grain approach was argued in Section 3.3 based on the poor worse case space and time requirements of existing fine grain memory models and the untenable development complexity of the static approach. Although the Entropy Hypothesis is not proven correct (and probably never will be due to the complexities of the tradeoffs involved), it motivates the direction that research is to take in order to address the disadvantages of these models. Using the detailed analysis of the disadvantages of existing fine grain solutions in Section 2.1, it was argued that a memory model for real-time and embedded systems must address the overheads of tracing and fragmentation. A coarse grain model achieves the latter by definition and makes the former a feasible option as the lifetime of a smaller number of elements must be explicitly defined. However, this does not address the problem of development complexity. The thesis hypothesis is carefully specified here as it states that a coarse grain model can *help* meet the challenge of achieving tight space and time overheads without overburdening the developer with memory concerns. The reason for this is that there is always the caveat that the underlying memory usage information is easy to express in the chosen model. This introduced the concept of *targeting* a memory model. Therefore a coarse grain model helps meet this challenge if object lifetime aggregation depending on some criteria is implicitly manifested in the application. If this is the case, then the memory model must provide a way of specifying the boundaries of this aggregation.

The coarse grain model investigated in this thesis is based on region boundaries that are defined by object lifetime aggregation against the control flow of the program. Although these boundaries could be expressed directly in program code, the separation of this concern can aid application development and maintenance. This concept is not new and is becoming a more popular software engineering technique [60]. The Region Partitioning Model developed in Chapter 4 provides a mechanism for describing this memory aspect. The advantages of this separation were demonstrated in the case study developed in Section 7.1.

Part 2: Taking advantage of information about the allocation and deallocation pattern of regions is known to provide further savings in both the temporal and spatial dimensions. However, it can be proven that the strictly scoped model as provided in the RTSJ fails under certain conditions with

memory requirements being infinite.

The second part of the thesis hypothesis was proven in Section 5.6 by using RPM$^{\circlearrowleft}$ to show how enforcing the lifetime of regions in a scoped order for a particular application will result in infinite memory requirements. This proof was based on a single-threaded application with a particular pattern of region lifetime. It was proven that an optimal scope ordering of this application for the specific case when the number of regions in the application is of size $n = 2^k$ will result in memory requirements at least of size $(k+1)\times \mid r_i \mid$. Therefore, if this lifetime pattern is present in a non-terminating application, then the memory requirements when adopting a scoped ordering would be infinite. This is because the memory requirements depend on the length of the application and the expression $(k+1)\times \mid r_i \mid$ does not converge to a constant.

Part 3: Java applications can target the scoped, coarse grain model of the RTSJ without incurring the claimed high development complexity. However, this requires more functionality to express object lifetime, better use of this information by implementations and a development approach suitable for targeting this model. Nevertheless, given the potential failings of a strictly scoped model, alternatives must be provided in the RTSJ.

Finally, the third part of the thesis hypothesis was proven by applying the RPM to a Java environment. Achieving this required two extensions to the RTSJ at the API-level in order to allow better expression of object lifetime; an extension in the specification of the requirements of virtual machine implementations in order to make better use of information expressed in the memory model; and a development approach that separates region lifetime information from program logic.

1. The first API-level extension, recyclable memory areas, addresses the failing of a strictly scoped ordering of regions as shown when proving the second part of the thesis hypothesis. This extension was introduced in Section 5.6.2. The second API-level extension, the separation of object lifetime from the object graph through Scoped Reference Objects, provides for a tighter specification of the true lifetime of objects, thereby reducing the worst case memory requirements. This extension was described in Section 6.1.2.

2. The advantages of a scoped ordering of regions is lost in current RTSJ implementations as memory subsystems do not allocate and deallocate the backing stores associated with scoped memory objects in a scoped order. This therefore introduces fragmentation that in the worst case can be larger than that experienced in a fine grain model. This problem was identified and addressed in Section 5.8 by proposing that the RTSJ make stricter requirements in the way memory subsystems are implemented, an example of which was also provided therein.

3. Finally, the abstraction provided by the RPM was used to define a separation between the memory concern of the application and program logic. It was shown in Section 7.1 how an application could be developed to target $\mathrm{RPM} - \mathrm{Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$ without considering memory particularities, (in particular the object graph) when developing the logic of regions. The decision of which coarse grain memory subsystem would ultimately be used could be left until deployment. If the chosen model were a scoped model, then an ordering of the scoped order of regions can be derived automatically using the Algorithms developed in Chapter 5.

7.4 Summary

The chapter demonstrated how an application that targets the $\mathrm{RPM} - \mathrm{Scoped}^{\mathrm{cRPM}^{\|\sigma\rho}}$ memory model is developed in Java. The separation of program logic from the expression of region lifetime as well as the separation of region lifetime from the object graph was shown to be important in simplifying development and maintenance. Next, a critique of the decision to adopt a scoped ordering for regions in a coarse grain model was provided. In particular, the advantages and disadvantages of this approach in comparison to a dynamic allocation policy based on existing offline analysis were discussed. It was concluded that as both approaches have benefits particular to the type of information known about the memory usage of the application, both approaches can provide a contribution as part of a memory subsystem. Finally the thesis hypothesis was reiterated in order to demonstrate how each of its three parts were proven.

Chapter 8

Conclusion

This chapter concludes this thesis by summarising the key contributions of this research. A number of limitations of this work are also identified. Future research directions are discussed, both in relation to the Region Partitioning Model and to the general memory management problem in real-time and embedded systems. Finally, a brief note on the key message this thesis tries to convey ends this work.

8.1 A Summary of the Key Thesis Contributions

The motivation for the work in this thesis is based on the argument that the traditional static memory model currently adopted in the development of real-time and embedded systems will increasingly prove unfeasible as these applications become larger and more complex. A dynamic memory model suited to this application domain can be adopted in order to reduce complexity as long as the overheads incurred by these models are not incommensurably large in comparison to the space and time requirements experienced in a static model.

Perhaps the most important contribution to the general area of research into dynamic memory management and the adaptation of this research to the specialised real-time domain is captured by the Entropy Hypothesis. The development of this hypothesis was based on an in-depth study of previous research into memory management in real-time and non real-time systems. This area of research now spans at least five decades and the analysis carried out in Chapter 2 captures only a small part of this research. The central argument behind the Entropy Hypothesis is essentially an intuitive one: as more information about a memory model is utilised, the space and time overheads are reduced. Based on this argument, the Entropy Hypothesis provides a unified model for describing the complex tradeoffs between space overheads, time overheads and the development complexity of different memory models. The arguments for and against real-time garbage collection, region-based models and other solutions are often subjective [5] and depend on which one of these three dimensions the advocate of that particular technology wishes to emphasise. The Entropy Hypothesis does not

prove any particular technology to be more suitable than another; what it does provide is a reference against which an argument in favour or against that technology can be made. Perhaps more crucially, the Entropy Hypothesis directs research away from refinement of memory models that simply shift overheads between the space and time domain and towards new memory models that allow the expression of known information about memory usage in the application as well as research into how this information can be used.

This thesis provides three key contributions for real-time Java application developers and researchers (including virtual machine developers). The first of these contributions which is of primary interest to RTSJ application developers is the proposed extension of the RTSJ reference semantics to incorporate scoped reference objects. The restrictions imposed by the scoped reference rules make development and maintenance of even the more simple applications incommensurably complex. Scoped reference objects provide a mechanism to eliminate these restrictions. The second key contribution is the detailed investigation into the effects of a scoped ordering of regions. The problems of describing patterns of region lifetime that follow a first-in-first-out order are well known but the proposed solutions are more akin to software "hacks". Mapping the scoped memory model to a memory pool model is an example of such a badly thought out solution [24]. The solution of recyclable memory areas on the other hand is derived by understanding exactly why scoped ordering fails. This investigation therefore enabled this thesis to propose a minimal extension to the RTSJ that addresses the source of this problem. Finally, the separation between the expression of lifetime information and the underlying memory subsystem isolates a key fallacy of RTSJ implementations: the failure of these implementations to take advantage of the scoped, coarse grain model. The memory subsystems proposed in this thesis are only initial ideas into how this model is best implemented; of more importance is that an under-explored area for future research has been uncovered.

Finally, the Region Partitioning Model (RPM) is itself an important contribution in that it provides a way of expressing one particular type of memory usage information: lifetime aggregation against program flow. Although the original reason for developing the RPM was to investigate the implications of adopting a scoped model, separating the expression of region lifetime from any particular memory subsystem can benefit the software development process. Once again, the importance of the limited model developed in this thesis is dwarfed by the potential for future research in extending the RPM to capture more information (such as variable sized regions and improved memory sharing to name just two), developing tools to simplify the process of expressing and manipulating this information and ultimately developing algorithms that use this information to reduce pessimism in the calculation of space and time analysis.

8.2 Limitations of this Work

The most important limitation of this work is the assumption that object lifetime aggregation based on program flow is indeed implicitly manifested in applications. This assumption is recognised throughout this thesis, particularly in the thesis hypothesis where it is stated that such a model can *help* address the challenges that a dynamic real-time memory model must address. This point was also discussed in Section 3.2 where the importance of "targeting" a memory model was emphasised in relation to the Entropy Hypothesis. The RTSJ is subject to this same limitation as it proposes only one memory model which must be utilised by any real-time application.[1] Therefore, both the research carried out in this thesis as well as the RTSJ assume that coarse grain aggregation against program flow is indeed implicit in programs. While previous research into generational garbage collectors [58] indicate that this is indeed the case, the explicit capturing of this information has only been done on smaller applications. The case study described in Section 7.1 is small enough to be designed around this model and therefore fails to capture the complex lifetimes that might exist in larger applications. Therefore, one of the key future research directions is to identify whether object lifetime aggregation against program flow is a phenomenon that occurs often enough to motivate such a model.

The RPM, its application to the scoped model, the proposed memory subsystems, and the implementation of Java classes to describe region lifetimes were all only developed in part. In every case, each of these four areas were developed until the point that they proved the relevant part of the thesis hypothesis. Therefore, despite the contributions described above, the utilisation of the results of this research in developing real-world applications would require several extensions to this work.

8.3 Future Work

There are four main directions for future work uncovered in this thesis. The first of these is captured by the arguments made in describing the Entropy Hypothesis: a memory model is suitable for a given application only in the degree to which it can capture information of memory usage in the application. The goal of future research must therefore lie in identifying this information and providing ways of allowing this information to be expressed in order for the underlying memory subsystem to make use of it. This is a significant shift from current research directions that deliver only marginal improvements due to the implicit assumption that expressing lifetime information implies unnecessary burdens on application developers.

The second direction for future work is the continued development of the Region Partitioning

[1] In fact, without the extensions proposed in this thesis, the RTSJ assumes even stricter rules on lifetime aggregation.

Model and its application to the scoped ordering of regions. A number of research directions in this area have been identified throughout this thesis and these can be grouped into four main categories:

- The extension of the RPM to allow the expression of more complex information about region lifetime; in particular, the model needs to be extended to allow the expression of regions sharing and synchronisation in programs with threads that have conditional and iterative control flows.
- Further research into how the information expressed in the RPM can be used by a memory subsystem and the implementation of these subsystems. In particular, the focus of this research should be into memory sharing in programs with conditional flows, region sharing and region synchronisation in both scoped and unscoped models.
- The development of tools to aid the process of expressing lifetime information, for example by depicting on a control flow graph where the correctness of a region's specified deallocation point fails.
- As noted by the first limitation described in Section 8.2, an investigation is required into whether object lifetime aggregation is a phenomenon exhibited implicitly in larger applications. The usefulness of the RPM and RTSJ models depend on this.

The approach taken in this thesis to investigate the effect of a scoped ordering of regions is based on the RPM and a transformation from unconstrained to constrained versions of the model. Existing offline analysis on the other hand maps the allocation problem to a graph colouring problem [67, 82] or equivalently, a strip-packing problem [15, 94]. The unique approach of shifting allocation and deallocation points could probably also be described as a special strip-packing problem with additional constraints on where strips can be placed. An equivalence between *annotateWithEIA()* and such an algorithm could then be shown.

Finally, it is noted that distributed systems that use Java's Remote Method Invocation (RMI) abstraction employ distributed garbage collection techniques that are usually based on reference counting. There has recently been a new initiative to incorporate real-time semantics into RMI in order to allow RTSJ-compliant applications to be distributed across a network while still providing real-time guarantees [123, 32, 34]. The biggest challenge is to understand how the RTSJ scoped memory model can be used to specify the lifetime of remote objects in the same way that it specifies the lifetime of ones local to it. This is a unique problem that is not addressed in previous work on real-time distributed object systems [101] as these generally assume that the lifetime of a remote object is controlled entirely by local processes. The region partitioning model can provide a solution to this problem by implementing region sharing semantics across the network. Client and server threads are therefore specified as separate RPM threads for which the lifetime of the remote object is defined by the lifetime of the region it resides in as per normal RPM semantics.

8.4 Final Word

The memory management problem in hard real-time systems is still somewhat of an emerging problem. A key contributor to this fact is the disparity between the development process of traditional non real-time software and that of real-time applications. In the development of traditional software systems, the advances in processor power and memory availability have kept abreast of the increasing performance requirements of these systems. Coupled with these requirements has been the necessity of suitable abstractions for the development of increasingly complex systems. The constraints of real-time and embedded systems has meant that many of these programming abstractions could not be adopted in these environments. Therefore, while non real-time developers make use of abstractions such as polymorphism and dynamic class loading in object models, real-time developers remain constrained to relatively low-level abstractions in order to provide the necessary guarantees. The world of the hard real-time developer therefore is to a large extent limited to using restrictive subsets of programming languages such as Ada's Ravenscar profile [43] and SPARK [17] subsets and low-level programming in C or assembly. This situation is gradually changing. The increasing complexity of real-time and embedded systems has driven research into bringing these abstractions to the real-time application domain without compromising the predictability of the system but while still achieving the performance requirements of these applications. This thesis has considered just one aspect of the research to bring abstractions readily available in traditional development environments to the real-time domain: the memory management problem. The work carried out in this thesis is a step towards understanding how to address this problem as well as providing a partial solution.

The real-time Java community still maintains widely diverging views as to which memory model should ultimately be adopted in the RTSJ. The focus on the memory management problem in current research into making Java suitable for real-time is testament to this. However, opinion converges on the understanding that the success of the RTSJ depends largely on developing a memory model suitable for developing increasingly complex real-time and embedded applications.

Appendix A

Algorithms for creating a CFG from a Program in *eRPM*

The algorithms in this appendix are used for creating the control flow graph of a program specified using eRPM notation. Algorithm 3 is passed a `BlockList` instance and calls Algorithm 4 with this instance. Algorithm 4 calls Algorithm 5 to build the set of edges and regions of the `BlockType` of this instance and then recursively calls itself to process the remaining `BlockList`.

Data : P:Program = ⟨B:BlockList⟩
Result : (R:Set of Region, E:Set of Edge, r_{start}:Region, r_{end}:Region)
R = ∅; E = ∅;
EndRegions: Set of Region;
r_{end} = new Region('r_{end}', SimpleRegion);
(R, E, r_{start}, EndRegions) = ProcessBlockList(B, ∅, ∅);
R = R ⋃{r_{end}} ;
forall L ε EndRegions **do**
    E = E ⋃ {(L, r_{end}, Forward, null, null)};
end

Algorithm 3: CFG(P:Program = ⟨B:BlockList⟩)

Data : B:⟨BlockList⟩, R:Set of Region, E:Set of Edge
Result : (R:Set of Region, E:Set of Edge, FirstRegion:Region, EndRegions:Set of Region)
r_{start}:Region = null;
FirstRegion = null;
LastList: Set of Region = ∅;
if B = ⟨b:BlockType⟩ **then**
    (R, E, FirstRegion, EndRegions) = ProcessBlock(b, R, E);
end
if B = ⟨b:BlockType⟩,⟨bl:BlockList⟩ **then**
    (R, E, FirstRegion, InitialEndRegions) = ProcessBlock(b, R, E);
    (R, E, NextFirstRegion, EndRegions) = ProcessBlockList(bl, R, E);
    forall L ε InitialEndRegions **do**
        E = E ⋃ {(L, NextFirstRegion, Forward, null, null)};
    end
end

Algorithm 4: ProcessBlockList(B:⟨BlockList⟩, R:Set of Region, E:Set of Edge

```
Data     : ⟨b:BlockType⟩, R:Set of Region, E: Set of Edge)
Result   : (R:Set of Region, E: Set of Edge, OutFirst:Region, OutLastList:Set of Region)
if b = ⟨S:SequenceBlock⟩ then
    OutFirst = null;
    OutLastList = ∅;
    LastRegion:Region = null;
    r:Region = null;
    while S ≠ null do
        if S = ⟨s:Region⟩ then
            r = new Region(toString(s), SimpleRegion);
            R = R ⋃{r};
            OutLastList = {r};
            S = null;
        end
        if S = ⟨s:Region⟩,⟨ss:SeqBlock⟩ then
            r = new Region(toString(s), SimpleRegion);
            R = R ⋃ {r};
            LastRegion = r;
            S = ss;
        end
        if LastRegion ≠ null then
            E = E ⋃ {(LastRegion, r,Forward, null, null)};
        else
            OutFirst = r;
        end
    end
end
if b = ⟨L:LoopBlock⟩ then
    if L = (ann = i:int | inf),⟨B:BlockList⟩ then
        (R,E,OutFirst,OutLastList) = ProcessBlockList(B,R, E);
        dummy:Region = new Region('dummy', SimpleRegion);
        R = R ⋃ {dummy};
        forall  I ε OutLastList do
            E = E ⋃ {(I, dummy, Forward, null, null)};
        end
        E = E ⋃ {(dummy, OutFirst, Back, null, ann)};
        OutLastList = {dummy};
    end
    if L = c:Condition, ⟨B:BlockList⟩, ann = max:int then
        r = new Region(toString(c), ConditionalRegion);
        dummy:Region = new Region('dummy', SimpleRegion);
        (R, E, OutFirst, OutLastList) = ProcessBlockList(B, R, E);
        forall  I ε OutLastList do
            E = E ⋃ {(I, dummy, Forward, null, null)};
        end
        E = E ⋃ {(dummy, r, Back, null, ann)};
        R = R ⋃ {r,dummy};
        E = E ⋃ {(r, OutFirst, Conditional, 1, null)};
        OutFirst = r;
        OutLastList = {r,dummy};
    end
end
if b = ⟨C:CondBlock⟩ then
    C = c:Condition,⟨b1:BlockList⟩,⟨b2:BlockList⟩
    OutFirst1:Region;
    OutLastList1:Set of Region;
    OutFirst2:Region;
    OutLastList2:Set of Region;
    r = new Region(toString(c), ConditionalRegion);
    (R, E, OutFirst1, OutLastList1) = ProcessBlockList(b1, R, E);
    (R, E, OutFirst2, OutLastList2) = ProcessBlockList(b2, R, E);
    R = R ⋃ {r};
    E = E ⋃ {(r, OutFirst1, Conditional, 1, null)};
    E = E ⋃ {(r, OutFirst2, Conditional, 0, null)};
    OutFirst = r;
    OutLastList = OutLastList1 ⋃ OutLastList2;
end
```

Algorithm 5: ProcessBlock(⟨b:BlockType⟩, R:Set of Region, E: Set of Edge)

Appendix B

Algorithms for *executeInArea()* and delayed allocation

The algorithms in this appendix are used for automatically creating a scoped ordering of regions for programs in RPM − Scoped$^{\text{cRPM}^{\rightarrow}}$. The functionality of these algorithms is described in Section 5.3.

Data : P: `Program` = $r_{\overrightarrow{1,n}}$: `RegionSequence`
Result : AnnP: `AnnotatedRegionSequence`
SSets=GenerateSSets(P)
AnnP=GetBestSequenceAnnotation($r_{\overrightarrow{1,n}}$)
return AnnP

Algorithm 6: annotateWithEIA(P: `Program`): Returns Best Annotation that minimises $MemReq_{LSL}^{RTSJ}(P)$

Data : Seq=$r_{\overrightarrow{i,j}}$: `RegionSequence`
Result : AnnSeq: `AnnotatedRegionSequence`
if *EDP(SSets[i][|SSets|])*=r_j^+ **then**
    /*$r_{\overrightarrow{i,j}}$ `is a chain` */
    AnnSeq=GetBestChainedSequenceAnnotation($r_{\overrightarrow{i,j}}$)
else
    /*$r_{\overrightarrow{i,j}}$ `is not a chain` */
    x: `Region`=RegionFromEDP(EDP(SSets[i][|SSets|]))
    chain: `RegionSequence`=$r_{\overrightarrow{i,x}}$
    y: `Region`=r_{x+1}
    AnnSeq=JoinChains(GetBestChainedSequenceAnnotation(chain), GetBestSequenceAnnotation($r_{\overrightarrow{y,j}}$))
end
return AnnSeq

Algorithm 7: GetBestSequenceAnnotation(Seq: `RegionSequence`): Returns Best Annotation for Region Sequence Seq

Data : Seq=$r_{i,j}^{\rightarrow}$: `RegionSequence`
Result : AnnSeq: `AnnotatedRegionSequence`
PossibleAnnotations: `Vector of AnnotatedRegionSequence`
for *count: int=1 to |SSets[i]|* **do**
  c: `AnnotatedRegionSequence`=Clone(Seq)
  Add **ENTER** in c at r_i^-
  Add **EXIT** in c at EDP(SSets[i][count])
  LastRegion: `Region`=RegionFromEDP(EDP(SSets[i][count]))
  EIASet: `Vector of Region`=$\{r_x;\ r_i < r_x <= LastRegion \wedge EDP(r_x) > EDP(SSets[i][count])\}$
  NextRegionSequence: `RegionSequence`=$r_{\overrightarrow{i+1,LastRegion}} - EIASet$
  if *(NextRegionSequence)* $\neq \varnothing$ **then**
    F: `AnnotatedRegionSequence`= GetBestSequenceAnnotation(NextRegionSequence)
    ReplaceAnnotation(c,NextRegionSequence,F)
  end
  if *(EIASet)* $\neq \varnothing$ **then**
    PossibleAnnotations += DoEIA(r_i,EDP(SSets[i][count]),c,1,EIASet)
  end
  AnnSeq=SelectBestAnnotation(PossibleAnnotations)
  return AnnSeq
end

Algorithm 8: GetBestChainedSequenceAnnotation(Seq: `RegionSequence`): Returns Best Annotation for Chained Region Sequence Seq

Data : start=r_i: `Region`
Data : last=r_j^+: `InterRegionLocation`
Data : c: `AnnotatedRegionSequence`
Data : nesting: `int`
Data : EIASet: `Vector of Region`
Result : AnnSeqVector: `Vector of AnnotatedRegionSequence`
P: `Vector of Region`=SortByEDP(EIASet)
r_p:Region=P[1];
TestForr_p: `Vector of Region`=SSets[p];
AnnSeqVector: `Vector of AnnotatedRegionSequence`
for *count: int=1 to |TestForr$_p$|* **do**
  currentEDP: `InterRegionLocation`= EDP(TestForr_p[count])
  newc: `AnnotatedRegionSequence`=Clone(c)
  Add **ENTER** in newc at r_i^- (outmost position if other **ENTER**s exist)
  Add **EXIT** in newc at currentEDP (outmost position if other **EXIT**s exist)
  ToWrap: `Vector of Region`=$\{r_x;\ r_x \epsilon EIASet \wedge EDP(r_x) <= currentEDP\}$
  $\forall\ r_y \epsilon ToWrap$, wrap in **EIA-ENTER/EIA-EXIT** with index *nesting* region r_y
  NewProblematicRegions: `Vector of Region`=$\{r_x;\ r_j^+ < r_x < currentEDP \wedge EDP(r_x) > currentEDP\}$
  r_{pcur}: `Region`=RegionFromEDP(currentEDP)
  RemainingAnnotation: `RegionSequence`=$r_{\overrightarrow{j+1,pcur}}$− NewProblematicRegions)
  if *(RemainingAnnotation)* $\neq \varnothing$ **then**
    F: `AnnotatedRegionSequence`= GetBestSequenceAnnotation(RemainingAnnotation)
    ReplaceAnnotation(newc,RemainingAnnotation,F)
  end
  RemainingProblematicRegions: `Vector of Region`= EIASet - ToWrap + NewProblematicRegions
  if *(RemainingProblematicRegions)* $\neq \varnothing$ **then**
    AnnSeqVector +=DoEIA(start, currentEDP, newc, nesting++, RemainingProblematicRegions)
  else
    AnnSeqVector +={newc}
  end
end
return AnnSeqVector

Algorithm 9: DoEIA(start: `Region`, last: `InterRegionLocation`, c: `AnnotatedRegionSequence`, nesting: `int`, EIASet: `Vector of Region`): Return Possible Annotations when applying *executeInArea()* on c

Data : r_i: `Region`
Result : S: `Vector of Region`
$S = r_i + \{r_x;\ r_i < r_x < EDP(r_i) \wedge EDP(r_x) > EDP(r_i)\}$
SortByEDP(S)
i=2
while $i <=\mid S \mid$ **do**
    $S = S + \{r_x;\ EDP(S[i-1]) < r_x < EDP(S[i]) \wedge EDP(r_x) > EDP(S[i])\}$
    SortByEDP(S)
    i++
end
return S

Algorithm 10: GenerateSSet(r_i: `Region`): Generating the set S for a Region r_i

Data : P: `Program` $= r_{\overrightarrow{1,n}}$: `RegionSequence`
Result : SSets: `Vector of Vector of Region`
foreach $r_i \epsilon P$ **do**
    SSets[i]=GenererateSSet(r_i)
end
return SSets

Algorithm 11: GenerateSSets(P: `Program`): Generating the Vector SSets

Appendix C

An Initial Investigation into Multi-threading in eRPM

This appendix explores how multi-threaded program with threads described in eRPM might have synchronisation and region sharing specified. Its purpose is to serve as a starting point for this extension to the RPM by investigating how these operations might be described and investigates the correctness criteria for the region sharing and synchronisation specifications. This appendix is included as it provides some useful insights into the complexities of incorporating these operations in models with non-sequential control-flows and also addresses some of the problems encountered in deriving the correctness criteria. The most important piece of further work that would be required is a derivation of the memory requirements of programs in this model. This would require the derivation of execution traces and memory traces as in cRPM$^{\|\sigma}$.

C.1 Describing Programs in eRPM$^{\|\sigma\rho}$

Recall that the constrained versions of RPM$^{\|\sigma}$ and RPM$^{\|\sigma\rho}$ (cRPM$^{\|\sigma}$ and cRPM$^{\|\sigma\rho}$ respectively) were different only in that they used cRPM$^{\rightarrow}$ rather than RPM$^{\rightarrow}$ in calculating the contribution to the memory requirements of some region by other regions in the same thread. Since the unconstrained models are subset of the constrained ones, eRPM$^{\|\sigma\rho}$ considers the constrained version of the model and, therefore, threads have RAPs and RDPs specified for each region. A program P in eRPM$^{\|\sigma\rho}$ is specified as $P = \langle T, r_m, D, H, Q, W \rangle$ where T is a set of threads as described in eRPM and H is a set of synchronisation points for regions that do not lie in infinite loops. Similar to region sharing in RPM$^{\|\sigma}$, D is used to describe region sharing in eRPM$^{\|\sigma\rho}$ for regions not in infinite loops. Q and W are used to describe synchronisation points and region sharing for regions in infinite loops and will be considered below. The specification of RAPs, EDPs and RDPs are identical to those for eRPM for allocation and deallocation points that fall in the same thread as the region under investigation. The specification of region sharing and synchronisation and the criteria

for correctness of the elements D, H, Q and W are defined next.

C.1.1 Specifying Region Sharing and Synchronisation in eRPM$^{\|\sigma\rho}$ without Infinite Loops

Consider first the subset of eRPM$^{\|\sigma\rho}$ which has no infinite loops in threads and for which Q and W are therefore empty. Also, assume for the time being that P has no synchronisation points (that is H is also empty) in order that region sharing described by the set D be considered alone. Recall that in cRPM$^{\|\sigma}$, region sharing is captured by identifying for some region a deallocation point in every thread. The RDP specification was therefore extended to allow this to be expressed. A similar process is required here. However, in contrast to cRPM$^{\|\sigma}$ where every thread is composed of only one track, eRPM programs can contain a number of tracks and not all regions are guaranteed to execute. Therefore, every region iteration must specify in its RDP not only one deallocation point for every thread but a set of deallocation points for every thread. Within this set of deallocation points would be inter-region locations that, as in single threaded versions of *eRPM*, some are guaranteed to be reached and others which are not. The correctness criteria for these deallocation points will be developed shortly and are similar as those for deallocation points in the other versions of the RPM developed so far. Again, the motivation is to ensure that a region is always guaranteed to be deallocated at some point.

Rather than extending the specification of RDPs as was done for cRPM$^{\|\sigma}$, a new element $\widetilde{\text{RDP}}^{\mathcal{M}^{\text{eRPM}^{\|\sigma\rho}}}(r_i)$ is introduced for every region that describes for each iteration of that region the deallocation points in other threads. The set composed of all these elements defines the set D. Let P have n threads. For some region r_i^j which identifies a region r_i in thread τ_j, $\widetilde{\text{RDP}}^{\mathcal{M}^{\text{eRPM}^{\|\sigma\rho}}}(r_i^j)$ is a set $\{e^1, e^2, \ldots, e^n\} - \{e^j\}$. The elements of this set contain one less element than the number of elements in T since for a thread τ^j, the regular EDPs and RDPs are used for deallocation points in this thread and therefore there is no element e^j. Every element e^k is specified in the same way as RDPs in eRPM, thereby identifying the deallocation points of every iteration of r_i^j in thread τ_k. For example, in Figure C.1 region r_2^1 in thread τ_1 can be deallocated after thread τ_2 has executed either r_4^2 or r_5^2 in the second iteration of M. Therefore, $\widetilde{\text{RDP}}^{\mathcal{M}^{\text{eRPM}^{\|\sigma\rho}}}(r_2^1) = \{\widetilde{\text{RDP}}(r_2^1, \{\}) = \{(r_4^{2^+}, \{M = 2\}), (r_5^{2^+}, \{M = 2\})\}\}\varepsilon D$.

The specification of the synchronisation set H is also similar to that for EDPs and RDPs in eRPM. For some region r_i^j which identifies a region r_i in thread τ_j, $H(r_i^j)$ is a set $\{e^1, e^2, \ldots, e^n\} - \{e^j\}$. The elements of this set contain one less element than the number of elements in T since for a thread τ^j, no synchronisation points from regions in τ^j to regions also in τ^j can exist and therefore there is no element e^j. Every element e^k is specified in the same way as RDPs in eRPM, thereby identifying the synchronisation points of every iteration of r_i^j in thread τ_k. For example, in Figure C.2, $H(r_2^1) =$

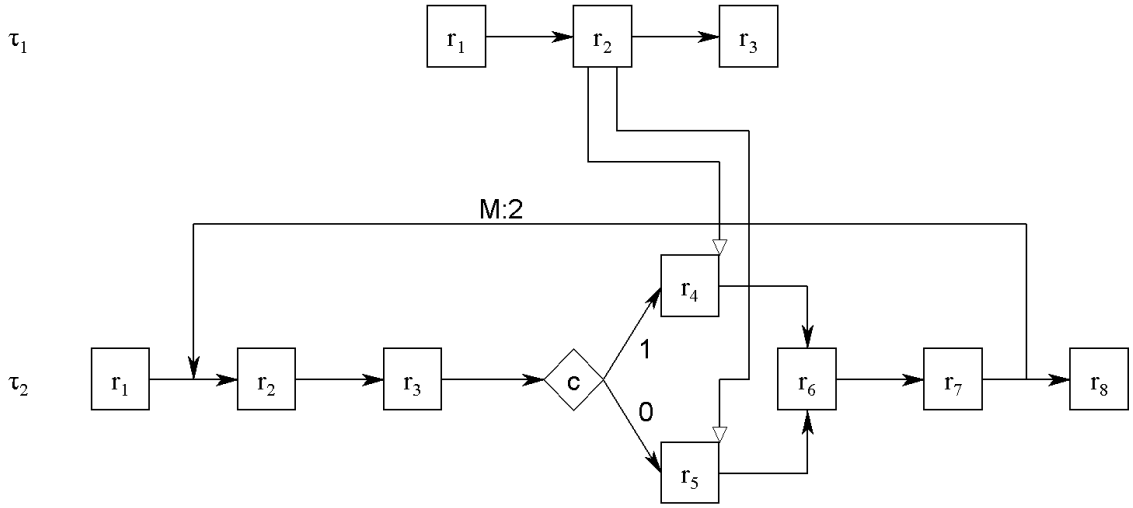


Figure C.1: Specifying Region Sharing in eRPM$^{\|\sigma\rho}$

$\{H(r_2^1,\{\}) = \{(r_4^2,\{M=2\}),(r_5^2,\{M=2\})\}\}\varepsilon H$.

C.1.2 Specifying Region Sharing and Synchronisation in eRPM$^{\|\sigma\rho}$ with Infinite Loops

Region sharing and synchronisation in programs in eRPM$^{\|\sigma\rho}$ with infinite loops must be handled with caution if it is to be guaranteed that finite memory requirements can be derived. Consider first a program with a thread that contains a fixed infinite loop. A finite number of regions in this loops can have deallocation points and synchronisation points in other threads by using the notation described above. However, if an infinite set of regions is to also have these points specified, notation similar to that used for describing allocation and deallocation points in cRPM$^{\circlearrowleft}$ must be defined. Two cases are considered in turn for a region in an infinite loop having a deallocation point in another thread: that deallocation point being in a fixed loop, conditional loop or no loop at all and that deallocation point being in an infinite loop. No consideration of infinite&conditional loops is made here but this can easily be extended. In the first case, deallocation points are specified in a similar way to the notation used above but with an inequality expression used to define the set of regions for which the deallocation point holds. For example, $\widetilde{\text{RDP}}^{\mathcal{M}^{\text{eRPM}^{\|\sigma\rho}}}(r_i^1) = \{\widetilde{\text{RDP}}(r_i^1,\{I > 0\}) = \{(r_j^{2^+},\{M=2\}),(r_k^{2^+},\{M=2\})\}\}$ means that all iterations of r_i^1 in the infinite loop I can be deallocated only after r_j^2 or r_k^2 in the second iteration of M in thread τ_2. Although these regions may be guaranteed to be reached, there is no guarantee of when they will be reached. Therefore, the memory requirements for this program can not be determined. In order to make the memory requirements deterministic, the progress of the infinite loop must be synchronised with the progress of τ_2. For example, if $((r_i^1,\{I=6\}),\{(r_j^{2^+},\{M=2\}),(r_k^{2^+},\{M=2\})\})\varepsilon H$ in this example, then τ_1

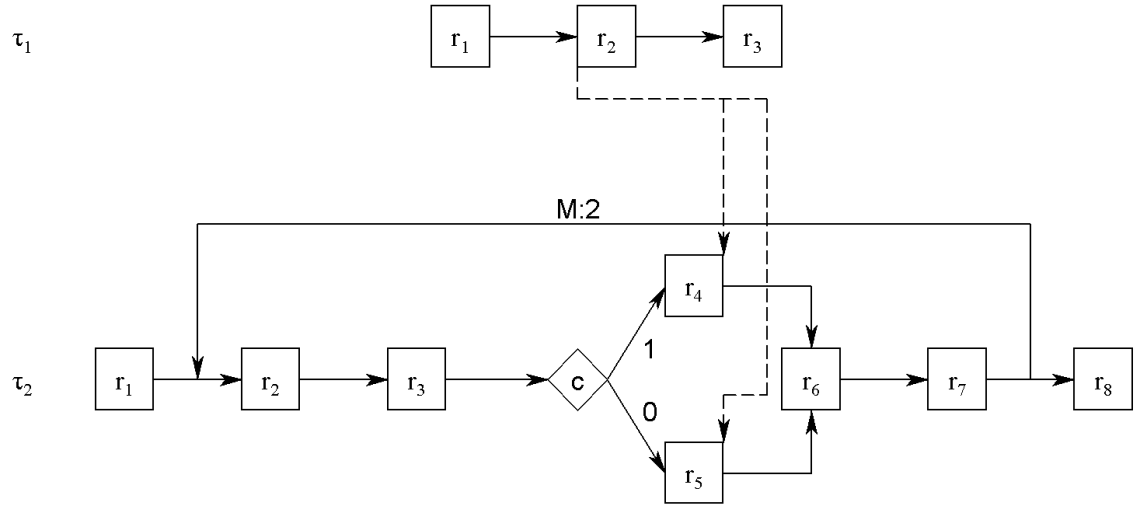


Figure C.2: Specifying Synchronisation in eRPM$^{\|\sigma\rho}$

can not progress beyond the 6^{th} iteration of r_i^1 before either the second iteration of r_j^2 or the second iteration of r_k^2 is reached. Since unrolling the loop I for 6 iterations and using this synchronisation point provides an equivalent result, this first case is disallowed.

Consider the second case for a region in an infinite loop having a deallocation point in another thread: that deallocation point also being in an infinite loop. Again, the progress of the thread for which the deallocation point is specified must be synchronised with the progress of the thread in which the deallocation point lies if the worst case memory requirements are to be determined. The set Q describes this synchronisation by showing how much progress the former thread (called the source thread) can make before having to wait for the latter (called the destination thread) as well as at which regions this pause must occur. The set W is used to describe region sharing specifically for regions in infinite loops. Q is a set of tuples, (S, I, J, N), where I and J are two infinite loops in two different threads, the source and destination threads respectively. N is an integer and S is a set with elements of the form (U, V). N identifies the maximum number of iterations that the source thread that may have to pause can be ahead of the other destination thread. The elements of S describe the synchronisation points themselves; U signifies where the source thread needs to pause and V signifies the points in the destination thread where the source thread may resume if sufficient iterations have passed. U identifies a single region and iteration and therefore takes the form of a pair $(r_{i'}^i, X)$ where X is the set of loops $r_{i'}^i$ is nested in (excluding I) and with the assigned iteration number. For each of these points, a number of regions in the destination thread are specified in V. Therefore, V is a set of elements of the form $(r_{j'}^j, Y)$ where Y is the set of loops $r_{j'}^j$ is nested in (excluding J) and with the assigned iteration numbers. For example, in Figure C.3, the source thread is specified to require a pause at r_5^i if either of r_4^2 or r_5^2 have not reached their

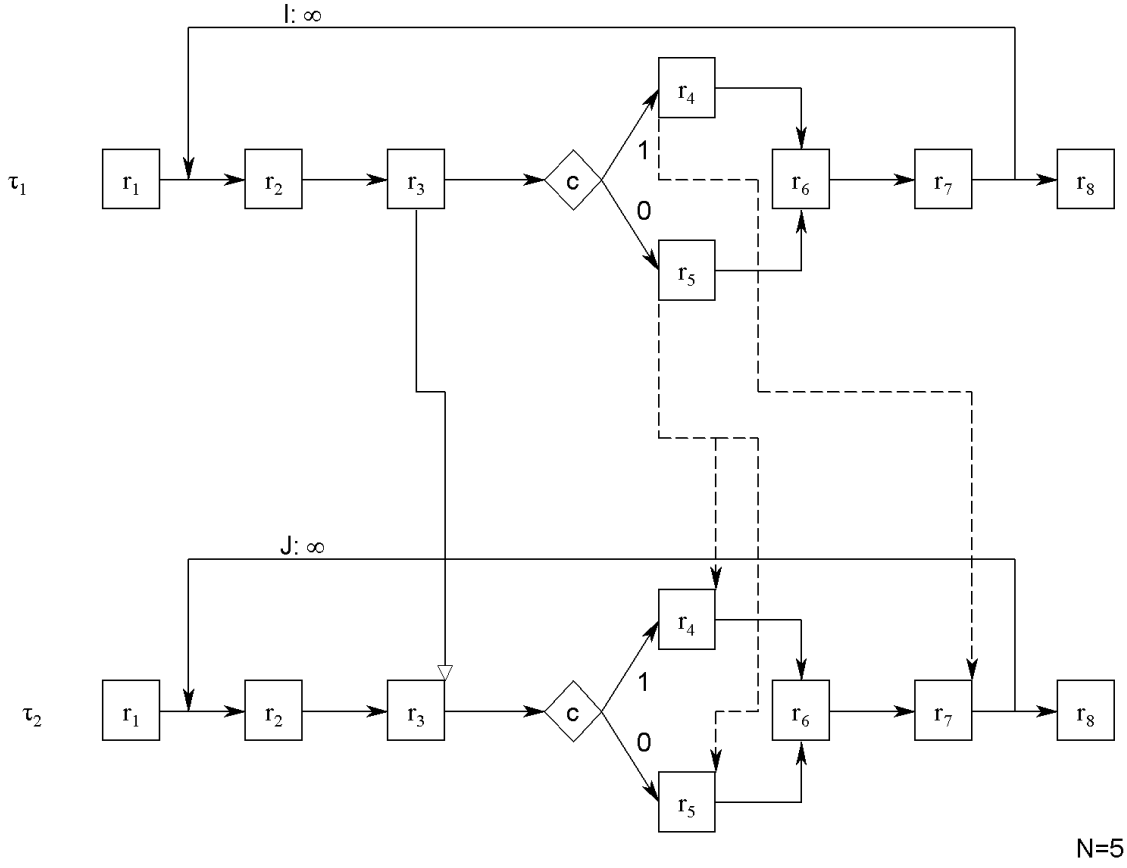


Figure C.3: Specifying Synchronisation in Infinite Loops in eRPM$^{\|\sigma\rho}$

5^{th} iteration in the destination thread. Similarly, thread τ_1 needs to pause at r_4^1 if r_7 has not been reached in the destination thread. In this example, the set Q will contain the element $(S, I, J, 5)$ where $S = \{(U_1, V_1), (U_2, V_2)\}$. U_1 is equal to $(\{\overline{c}\}, r_5^1, \{\})$ and U_2 is equal to $(r_4^1, \{\})$. Finally, the set V_1 is equal to $\{(r_4^2, \{\}), (r_5^2, \{\})\}$ and the set V_2 is equal to $\{(r_7^2, \{\}), (r_7^2, \{\})\}$.

There are several ways in which cross-region sharing could be specified in the set W. The more complex the chosen solution, the more information about memory usage could be specified and therefore, by the Entropy Hypothesis, the lower the memory and time overheads that are incurred. A simple example is chosen here that would suffice for the majority of programs. If this is not the case then a new way of specifying cross-region sharing would need to be specified and the correctness criteria and memory requirements derived. For simplicity, it is assumed that if memory for a region in an infinite loop in the source thread can not be freed before some point in the destination thread, then this will hold for all subsequent iterations of the source and destination threads. For example, recall that for the program in Figure C.3, the loop in τ_1 could never progress further than 5 iterations in the loop I beyond that of the loop J. Therefore, if it is specified that the first iteration of r_i^1

in infinite loop I can not be deallocated until the third iteration of r_j^1 in infinite loop J , then it also the case that the second iteration of r_i^1 in infinite loop I can not be deallocated until the fourth iteration of r_j^1 in infinite loop J, and so on. In general, the set W is made up of elements of the form (S, I, J, m, n), where I and J are two infinite loops in two different threads, the source and destination threads respectively. m and n are integers that identify the iteration numbers for the infinite loops I and J respectively. S is of the form (U, V) where U identifies a single region and iteration and therefore takes the form of a tuple $(r_{i'}^i, X)$. X is the set of loops $r_{i'}^i$ is nested in (excluding I) and with the assigned iteration numbers. For each of these points, a number of regions in the destination thread are specified in V. Therefore, V is a set of elements of the form $(r_{j'}^j, Y)$ where Y is the set of loops $r_{j'}^j$ is nested in (excluding J) and with the assigned iteration numbers. U identifies the region in the source thread whose M^{th} iteration in I can not be deallocated until one of the regions in the destination thread identified by V is reached at the N^{th} iteration of J. For example, if $(((r_3^1, \{\}), \{(r_4^{2+}, \{\}), (r_5^{2+}, \{\})\}), I, J, 1, 4) \varepsilon W$ for the program in Figure C.3, then the first iteration of r_3^1 in I can be deallocated only after the fourth iteration of J has reached the inter-region locations r_4^{2+} or r_5^{2+}. Similarly, the second iteration of r_3^1 in I can be deallocated only after the fifth iteration of J has reached the inter-region locations r_4^{2+} or r_5^{2+}, and so on.

C.2 Correctness Criteria for Programs in $\mathrm{eRPM}^{\|\sigma\rho}$

The correctness criteria for Programs in $\mathrm{eRPM}^{\|\sigma\rho}$ is based on correctness of RAP, EDP and RDP specifications for the allocation and deallocation points in the same thread as the region under investigation as well as correctness of the elements D,H,Q, and W.

C.2.1 Correctness of Region Sharing Specifications in $\mathrm{eRPM}^{\|\sigma\rho}$ with no Infinite Loops

The correctness criteria for the element D for programs in $\mathrm{eRPM}^{\|\sigma\rho}$ is based on the correctness of every sharing specified in the elements of D. Every element in the source thread must have exactly one guaranteed deallocation point for every track in the destination thread. Therefore, the correctness of $\widetilde{\mathrm{RDP}}^{\mathcal{M}^{\mathrm{eRPM}\|\sigma\rho}}(r_i^j)$ for every region execution of r_i^j in the program is found by checking whether $\mathrm{RDP}^{\mathcal{M}^{\mathrm{eRPM}}}(r_0^k) = e^k$ is correct for every element e^k in $\widetilde{\mathrm{RDP}}^{\mathcal{M}^{\mathrm{eRPM}\|\sigma\rho}}(r_i^j)$. r_0^k is a new region inserted before the first region in thread τ_k and nested in the same loops as r_i^j. If no sharing with a thread τ_k occurs for some iteration of r_i^j, then the deallocation point is set at ${r_0^k}^+$ for these iterations of r_0^k.

C.2.2 Correctness of Synchronisation Specifications in eRPM$^{\|\sigma\rho}$ with no Infinite Loops

The correctness criteria for the element H for programs in eRPM$^{\|\sigma\rho}$ is based on the correctness of every synchronisation point specified in the elements of H. The criteria for correctness are identical to those for the set D but with the extra requirement that synchronisation that can lead to deadlock as well as redundant synchronisation points are removed from H.

C.2.3 Correctness of Synchronisation Specifications in eRPM$^{\|\sigma\rho}$ with Infinite Loops

Recall that the set Q is a set of pairs, (S, I, J, N). A number of sanity checks can be carried out here. For example, it might be shown that the regions for which the synchronisation holds are indeed in the infinite loops I and J. A number of checks similar to the correctness criteria for deallocation points in eRPM can also be carried out to ensure the element S is valid. This basically involves ensuring that a guaranteed synchronisation point is reachable for every track in the infinite loop. If S is a set with elements of the form (U, V) then the specification is correct if every track in the infinite loop has exactly one region in V that is guaranteed to execute for every possible track through the infinite loop.

Appendix D

The Semantics of Reference Objects in the RTSJ

This chapter proposes semantics for applications using classes in the reference object classes of the `java.lang.ref` package in an RTSJ scoped memory environment.

D.1 Reference Objects in Java

Java defines the *reachability* of an object based on the type of reference an application holds to that object. An object is *strongly reachable* if it can be reached without traversing any reference object. It is *softly reachable* if it is not strongly reachable but can be reached by traversing a soft reference. It is *weakly reachable* if it can be reached by traversing a weak reference but it is not strongly or softly reachable. Finally, it is *phantom reachable* if can be reached by traversing a phantom reference but is not strongly, softly or weakly reachable. Otherwise, an object is said to be *unreachable*. The three types of reference objects in Java are described next in more detail:

Soft References

Soft references are cleared at the discretion of the garbage collector in response to memory demand. When low on memory, the garbage collector may choose to clear atomically all soft references to a softly reachable object and all soft references to any other softly-reachable objects from which that object is reachable through a chain of strong references. The referent is marked for finalisation and the referent's memory may be reclaimed when finalisation is complete. At that point or at a later time, the cleared reference objects registered with a reference queue are enqueued. Soft references are used for implementing memory sensitive caches.

Weak References

When the garbage collector determines that an object is weakly reachable, it will atomically clear all weak references to that object and all weak references to any other weakly reachable objects

from which that object is reachable through a chain of strong and soft references. It will declare the referent finalisable and at the same time or at some later time it will enqueue those newly-cleared weak references that are registered with reference queues. As with soft references, since weak reference objects are cleared automatically by the garbage collector, the referent's memory may be reclaimed when finalisation is complete. Weak references are used for implementing canonicalising mappings.

Phantom References

Phantom references differ from soft and weak references in two key ways. Firstly, invoking *get()* on an instance of `PhantomReference` always returns `null`. Therefore, phantom references must be registered with a reference queue to be useful. Secondly, phantom references are not automatically cleared by the garbage collector as they are enqueued. At this stage, the referents are only marked for finalisation. An object that is reachable via phantom references will remain so until all such references are cleared or themselves become unreachable. Therefore, though finalised, the memory of a phantom reachable object is not reclaimed until all phantom references are cleared. For this reason, at the end of the post-finalisation cleanup code, the program should call the *clear()* method on the phantom reference object to set the reference field of the reference object to `null`, allowing the garbage collector to reclaim the referent's memory. Phantom references are used for implementing pre-mortem cleanup actions in a more flexible way than is possible with finalisation

D.2 Reference Objects in the RTSJ

This sections discusses how the use of noheap threads that may interrupt the garbage collector places implementation concerns on two key virtual machine functions when using reference objects: clearing of reference objects and enqueuing on the registered reference queue. It is also shown how and when heap and noheap threads may use *get()* and process reference queues. The notation used will be A $\rightarrow$ B to signify that the reference object in memory area A references the referent in memory area B.

Heap $\rightarrow$ Heap: The same reachability semantics apply as in regular Java and the use of *get()* and the clearing of references objects is also the same for all reference object types.

The reference queue to which a reference object in heap is registered can be created in heap or immortal memory. In either case, processing of the reference queue should be done by a heap-using thread. Therefore, enqueuing of the reference object on a reference queue in immortal memory may leave the queue in an inconsistent state when interrupted by a noheap

thread. Nevertheless, an implementation would probably still use an appropriate enqueue algorithm that leaves the queue in a consistent state at each step of the enqueue operation and throw a `MemoryAccessError` in the usual way if a reference object in heap is dequeued by a noheap thread.

Immortal → Heap: The semantics here are again the same as in regular Java for reachability, though the use of *get()* differs in compliance with noheap thread usage. A new situation arises here in which the garbage collector modifies immortal memory. This occurs when soft and weak reference objects are cleared by the virtual machine. Therefore, it is necessary to ensure that clearing reference objects does not incur penalties on noheap threads. Even if clearing is done atomically, this would incur priority inversion, albeit bounded, on the noheap thread by the garbage collector for the duration of the atomic operation. A better solution is therefore to automatically fail with a `MemoryAccessError` when calling *get()* from a noheap thread, even if the result of this would have been `null`. The latter solution means that reference objects can be safely left in a semi-cleared state as long as they are fully cleared before any other heap-using thread executes. For cases requiring atomic clearing of multiple soft and weak references, both solutions are still valid as heap threads see an atomic clearing of the relevant reference objects.

Reference objects in immortal memory may be registered with reference queues in immortal or heap memory.[1] Note that a noheap thread can interrupt the garbage collector's enqueuing thread at any point and require access to the reference queue. This requires the enqueue operation to be wait-free as discussed in Section D.3.3. Reference objects in immortal memory can always be enqueued onto a reference queue, even if they are unreachable as they are guaranteed not to have been collected.

Scoped → Heap: Again, all three types of reference objects may be created with the same reachability semantics. The rules for using *get()* and clearing reference objects are the same as for Immortal → Heap.

Reference objects point to the reference queues they are registered with. Once their reachability changes, it is the reference queue that points to the reference object. This implies that reference objects can only be registered with a reference queue in the same scoped memory area. Enqueuing of reference objects must be done as described for Immortal → Heap. Note that all referent objects created in scoped memory together with their reference queues are lost when the scoped memory area is removed. These semantics are the same as when a reference object is itself not strongly reachable and is garbage collected in standard Java.

[1] The latter option is unavailable for noheap threads.

{Heap, Immortal, Scoped} → Immortal: Fails for all reference types, throwing an exception. Immortal memory is never to be subject to garbage collection, meaning that objects are never to be marked for finalization. Though the garbage collector scans immortal memory and may therefore identify objects that are not strongly reachable, it can only be assumed that, in the most general case, the garbage collector algorithm considers all objects in immortal memory to be root objects and therefore strongly reachable.

D.2.1 References to Scoped Objects

It would be expected that the creation of reference objects to scoped referents would be governed by the same rules as those of the RTSJ. It will be shown that this is not the case and some benefit can be derived by allowing phantom references to be created for referents in shorter-lived memory areas. The reachability semantics are different from regular Java as reachability of a referent now depends on the scoped memory reference count and not on the reachability of the referent. Each reference type is investigated in turn in order to identify how they may be used in this context.

Heap → Scoped: Referents of soft references are removed in response to memory requirements. However, even if a scoped memory region runs out of space, identifying softly reachable objects requires computation time. Doing this is contrary to the rationale for having memory areas that are not garbage collected in the first place. It may be argued that weak references could be used but must be done so with care as invoking *get()* would fail if the returned reference is assigned in a way that violates RTSJ rules. However, weakly reachable objects reachable by strong references from a cleared weak reference need to be cleared automatically and atomically by the garbage collector. These could exist in heap, immortal memory or the current and outer nested scopes. This would require immortal and scoped memory to be locked, potentially leading to noheap threads experiencing priority inversion due to the garbage collector. It is therefore more appropriate to only allow phantom references to be used to reference scoped objects as these are not cleared by the garbage collector. Clearing of the reference object by the application becomes redundant as the referents are reclaimed when the scoped area is cleared. Moreover at the time of enqueuing, the referent may have already been reclaimed. Therefore, it is not possible to determine whether pre-mortem or post-mortem cleanup would take place.

The reference queue can be in either heap or immortal memory. The garbage collector must be able to identify whether the scoped area of a phantom reference has been cleared in order to enqueue that reference object. The rules for enqueuing in the virtual machine and processing of the queue by the application are the same as for Heap → Heap.

Immortal → Scoped: As for Heap → Scoped above, it is proposes that only phantom references are allowed and that these retain the same reachability semantics. The rules for enqueuing and processing of the queue by the application are the same as for Immortal → Heap.

Scoped → Scoped: There are four possible scoped memory locations for the referent of a scoped reference object. The first is when the referent is in an outer scope. The new reachability semantics of scoped reference objects makes this option useless. Reachability changes of the referent can only be detected when the scope memory holding that referent is cleared. As this point the reference object and reference queue would also have been cleared. The same holds for the second case where a referent's reference exists in the same scoped memory area. In the last two cases the referent is either in an inner scope or in a scoped memory area outside the current scope stack. Here only allow phantom references are allowed for the same reasons given for Heap → Scoped references above.

As for Scoped → Heap references, reference queues can only be in the same scoped memory as the reference object. Enqueuing and processing of reference objects is the same as for Heap → Scoped references above.

D.3 Real-Time Reference Object Processing

Reference queues must be processed by an application in order to carry out the necessary logic when the reachability of the reference object changes. It is impossible to guarantee when a reference object is enqueued onto a reference queue due to the unpredictability of when the garbage collector runs or, for scoped reference objects, when the scope is cleared. However, the processing of reference objects can be done in a timely fashion using RTSJ `Schedulable`s. Note that scoped reference objects may also be registered with reference queues and appended to the queue at some point after the reference count of the scoped memory holding the referent reaches is cleared (or deallocated in the RPM). Typically this operation would be carried out at the time the scoped region is to be reclaimed and the finalizers of objects in the scope are run. Once again as there is no guarantee when regular reference objects are enqueued once deemed unreachable by the garbage collector, there is also no guarantee when scoped reference objects are enqueued. Each scoped memory area should maintain a list of scoped reference objects it needs to append to the reference queue once it is cleared. Scoped reference objects in heap can be assumed to always be strongly reachable until the referent's scope is cleared and, therefore, never subject to garbage collection.

D.3.1 Using Reference Queues

The `ReferenceQueue` class provides three methods to extract reference objects from a reference queue: *poll()*, *remove()* and *remove(int timeout)*. The first method is a non-blocking call that returns `null` if no reference object is enqueued. The latter two methods are blocking calls, the latter of which waits for a specified number of milliseconds. An application would typically use *poll()* and *remove(int timeout)* if no dedicated thread is to be assigned to the processing of the reference queue. If a dedicated thread is to be assigned to reference queue processing, *remove()* is called in the processing loop, blocking until an object is available in the queue. A reference queue may be processed by any number of threads, each calling any of these three methods.

The timeout parameter for *remove()* does not have real-time characteristics as a direct repercussion of the absence of real-time functionality in Java. Typically, the timeout is implemented using a *wait(int timeout)* operation on a lock that must be obtained before any enqueue or dequeue operation on the queue instance. If a reference object is enqueued by calling *enqueue()*, *notify()* is invoked to wake up an arbitrary thread blocked on one of the *remove()* methods.

D.3.2 Processing Reachability Change in Real-Time

Having multiple threads to service the same reference queue using blocking calls makes predictability hard. There are two solutions to this problem. The first is to associate just one `RealtimeThread` with assigned scheduling and releases parameters to one or more reference queues. Assuming a periodic thread is used, *poll()* is invoked at every period, servicing one reference object from a selected queue each time. In this case, the worst case execution time would be the greatest of the worst case times to process any of the reference queue objects.

An more attractive solution for notifying an application of reachability change of a reference object is to make use of the RTSJ's asynchronous event handler functionality. However, as a result of the data-less association between events and their handlers, unnecessary complexity is introduced unless only one handler is assigned to each reference object. Therefore, a more appropriate solution is to associate an event directly with a reference object, rather than with a reference queue. A new abstract class `RealtimeReference` is defined that extends `Reference`. Constructors of subclasses of `RealtimeReference` may take an `AsyncEvent` instance that may be associated with a set of `AsyncEventHandler`s in the usual way. When the reference object is cleared, the event is fired. Associating a unique event to each reference object will allow the application to identify which reference object has been cleared.

D.3.3 Implementing Reference Queues for Real-Time

Even with the above two solutions, it may be necessary to allow multiple threads to access a reference queue. In fact, reference queues are objects that are somewhat unique in that they are accessed and mutated by both the virtual machine and the application. An implementation would typically use a lock for the queue on which threads synchronise before adding or removing items from the queue. Note that the enqueue operation could be interrupted by a noheap thread. In particular, this could lead to a situation in which a noheap thread may have to wait for the garbage collector's enqueuing thread to complete its operation, leading to priority inversion caused by the garbage collector. Although this inversion may be bounded, it is still undesirable. Therefore, it is best to implement the queue without synchronisation on locks. The RTSJ provides this functionality in the form of wait-free queues. In particular, the reference queue can be implemented as a `WaitFreeWriteQueue` with an unsynchronised and nonblocking *write()* and a synchronised and blocking *read()*. The enqueue thread will fail to enqueue if interrupted by a noheap thread and attempt to enqueue again some time later.

Bibliography

[1] TimeSys' Real-Time Specification for Java Reference Implementation is avaliable at: http://www.timesys.com/rtj.

[2] Information on the Golden Gate project can be found at: http://research.sun.com/projects/dashboard.php?id=80.

[3] The Reference Object API for Sun's JDK 1.3 is avaliable at: http://java.sun.com/j2se/1.3/docs/guide/refobs/.

[4] *Microsoft C# Language Specifications*. Microsoft Press, Redmond, WA, USA, 2001.

[5] November 2003. Panel Session at the Workshop on Java Technologies for Embedded Systems (JTRES) 2004.

[6] Project Mackinac: Sun's Real-time Java Platform. Available at: http://research.sun.com/projects/mackinac/mackinac_whitepaper.pdf.

[7] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[8] Aiken, A., Fähndrich, M., and Levien, R. Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages. *SIGPLAN Notices 30*, 6 (1995), 174–185.

[9] Ajtai, M. Determinism versus Nondeterminism for Linear Time RAMs with Memory Restrictions. *Journal of Computer Systems Science 65*, 1 (2002), 2–37.

[10] Albus, J. S., McLean, C. R., Barbera, A. J., and Fitzgerald, M. L. Hierarchical Control for Robots and Teleoperators. In *Proceedings of the IEEE Workshop on Intelligent Control* (Albany, NY, USA, August 1985), G. Saridis and A. Meystel, Eds., pp. 39–49.

[11] Audsely, N. C. Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times. Tech. Rep. YCS-1991-164, University of York, Department of Computer Science, December 1991.

[12] Bacon, D., Cheng, P., and Rajan, V. T. A Real-Time Garbage Collector with Low Overhead and Consistent Utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)* (New Orleans, LO, USA, January 2003), ACM Press, pp. 285–298.

[13] Bacon, D., Cheng, P., and Rajan, V. T. The Metronome: A Simpler Approach to Garbage Collection in Real-Time Systems. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '03)* (Catania, Sicily, Italy, November 2003), R. Meersman and Z. Tari, Eds., vol. LNCS 2889 of *Lecture Notes in Computer Science*, Springer, pp. 466–478.

[14] Bacon, D. F., Cheng, P., and Rajan, V. T. Controlling Fragmentation and Space Consumption in the Metronome, a Real-Time Garbage Collector for Java. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, Compiler, and Tool for Embedded Systems (LCTES 03)* (San Diego, CA, USA, June 2003), ACM Press, pp. 81–92.

[15] Baker, B. S., and Schwarz, J. S. Shelf Algorithms for Two-Dimensional Packing Problems. *SIAM Journal of Computing 12*, 3 (1983), 508–525.

[16] Baker, H. G. List Processing in Real Time on a Serial Computer. *Communications of the ACM 21*, 4 (1978), 280–294.

[17] Barnes, J., and Barnes, J. G. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[18] Barrett, D. A., and Zorn, B. G. Using Lifetime Predictors to Improve Memory Allocation Performance. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming Language Design and Implementation (PLDI '93)* (Albuquerque, NM, USA, June 1993), ACM Press, pp. 187–196.

[19] Basanta-Val, P., García-Valls, M., and Estévez-Ayres, I. No Heap Remote Objects: Leaving Out Garbage Collection at the Server Side. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '04)* (Agia Napa, Cyprus, October 2004), R. Meersman, Z. Tari, and A. Corsaro, Eds., vol. LNCS 3292 of *Lecture Notes in Computer Science*, Springer, pp. 359–370.

[20] Beame, P. A General Sequential Time-Space Tradeoff for Finding Unique Elements. *Society for Industrial and Applied Mathematics (SIAM) Journal of Computing 20*, 2 (1990), 270–277.

[21] Beebee, W. S. Region-Based Memory Management for Real-Time Java. Master's thesis, Massachusetts Institute of Technology, September 2001.

[22] Beebee, W. S., and Rinard, M. C. An Implementation of Scoped Memory for Real-Time Java. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT '01)* (London, UK, 2001), vol. LNCS 2211 of *Lecture Notes In Computer Science*, Springer-Verlag, pp. 289–305.

[23] Belady, L. A. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal 5*, 2 (1966), 78–101.

[24] Benowitz, E., and Niessner, A. A Patterns Catalog for RTSJ Software Designs. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '03)* (Catania, Sicily, Italy, November 2003), Z. T. e. a. Robert Meersman, Ed., vol. LNCS 2889 of *Lecture Notes in Computer Science*, Springer, pp. 497–506.

[25] Benowitz, E., and Niessner, A. Experiences in Adopting Real-Time Java for Flight-Like Software. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '03)* (Catania, Sicily, Italy, November 2003), Z. T. e. a. Robert Meersman, Ed., vol. LNCS 2889 of *Lecture Notes in Computer Science*, Springer, pp. 490–496.

[26] Berger, E. D., Zorn, B. G., and McKinley, K. S. Reconsidering Custom Memory Allocation. *SIGPLAN Notices 37*, 11 (2002), 1–12.

[27] Blanchet, B. Escape Analysis for Java(TM). Theory and Practice. *ACM Transactions on Programming Languages and Systems 25*, 6 (November 2003), 713–775.

[28] Boehm, H. J. Memory Allocation Myths and Half-Truths. Available at: http://www.hpl.hp.com/personal/Hans_Boehm/gc/myths.ps.

[29] Boehm, H.-J., and Weiser, M. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience 18*, 9 (1988), 807–820.

[30] Bollella, G., Canham, T., Carson, V., Champlin, V., Dvorak, D., Giovannoni, B., Indictor, M., Meyer, K., Murray, A., and Reinholtz, K. Programming with Non-Heap Memory in the Real-Time Specification for Java. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)* (Anaheim, CA, USA, 2003), ACM Press, pp. 361–369.

[31] Bollella, G., and Gosling, J. The Real-Time Specification for Java. *Computer 33*, 6 (2000), 47–54. Available at: http://www.rtj.org.

[32] Borg, A. A Real-Time RMI Framework for the RTSJ. Tech. Rep. YCS-2003-365, University of York, 2003.

[33] Borg, A. On The Development of Dynamic Real-Time Applications in the RTSJ - A Model for Expressing Dynamic Memory Requirements . Tech. Rep. YCS-2004-379, University of York, June 2004.

[34] Borg, A., and Wellings, A. J. A Real-Time RMI Framework for the RTSJ. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS '03)* (Porto, Portugal, July 2003), Euromicro, IEEE.

[35] Borg, A., and Wellings, A. J. Reference Objects for RTSJ Memory Areas. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '03)* (Catania, Sicily, Italy, November 2003), R. Meersman and Z. Tari, Eds., vol. LNCS 2889 of *Lecture Notes in Computer Science*, Springer, pp. 397–410.

[36] Borg, A., and Wellings, A. J. Towards an Understanding of the Expressive Power of the RTSJ Scoped Memory Model. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '04)* (Agia Napa, Cyprus, October 2004), R. Meersman, Z. Tari, and A. Corsaro, Eds., vol. LNCS 3292 of *Lecture Notes in Computer Science*, Springer, pp. 315–332.

[37] Borg, A., and Wellings, A. J. Real-Time Memory Management: Life and Times. In *To Appear in Proceedings of the 18th Euromicro Conference on Real Time Systems (ECRTS '06)* (July 2006), Euromicro, IEEE.

[38] Borg, A., and Wellings, A. J. Scoped, Coarse-Grain Memory Management and the RTSJ Scoped Memory Model in the Development of Real-Time Applications. *To Appear: International Journal of Embedded Systems* (2006).

[39] Borg, A., Wellings, A. J., Gill, C., and Cytron, R. K. Real-Time Memory Management: Life and Times. Tech. Rep. WUCSE-2006-4, University of Washington in St. Louis, St. Louis, MO, USA, January 2006.

[40] Borodin, A., and El-Yaniv, R. *Online Computation and Competitive Analysis*. Cambridge University Press, New York, NY, USA, 1998.

[41] Boyapati, C., Salcianu, A., Beebee, W., and Rinard, M. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)* (San Diego, CA, USA, June 2003), ACM Press, pp. 324–337.

[42] Brooks, R. A. Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP '84)* (Austin, TX, USA, 1984), ACM Press, pp. 256–262.

[43] Burns, A., Dobbing, B., and Vardanega, T. Guide for the use of the Ada Ravenscar Profile in High Integrity Systems. Tech. Rep. YCS-2003-348, University of York, January 2003.

[44] Campbell, D. M. The Computation of Catalan Numbers. *Mathematics Magazine 57*, 4 (September 1984), 195–208.

[45] Chapman, R. Industrial Experience with SPARK. *Ada Letters XX*, 4 (2000), 64–68.

[46] Child, J. Real-Time Java Takes Aim at Embedded Control. Available at: http://www.rtcmagazine.com/home/article.php?id=100111.

[47] Chin, W.-N., Craciun, F., Qin, S., and Rinard, M. Region Inference for an Object-Oriented Language. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)* (Washington DC, USA, June 2004), ACM Press, pp. 243–254.

[48] Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V. C., and Midkiff, S. Escape analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)* (Denver, CO, USA, November 1999), ACM Press, pp. 1–19.

[49] Christiansen, M. V., and Velschow, P. Region-Based Memory Management in Java. Master's thesis, University of Copenhagen, May 1998.

[50] Corsaro, A. *Techniques and Patterns for Efficient Real-Time Middleware.* PhD thesis, Washinton University at St. Louis, December 2004.

[51] Corsaro, A., and Cytron, R. K. Efficient Memory-Reference Checks for Real-Time Java. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tools for Embedded Systems (LCTES '03)* (San Diego, CA, USA, June 2003), ACM Press, pp. 51–58.

[52] Corsaro, A., and Santoro, C. Design Patterns for RTSJ Application Development. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '04)* (Agia Napa, Cyprus, October 2004), R. Meersman, Z. Tari, and A. Corsaro, Eds., vol. LNCS 3292 of *Lecture Notes in Computer Science*, Springer, pp. 394–405.

[53] Corsaro, A., and Santoro, C. Optimizing JVM Object Management Operations to Improve WCET Predictability. In *Proceedings of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis* (Catania, Sicily, Italy, June 2004).

[54] Corsaro, A., and Schmidt, D. C. The Design and Performance of the jRate Real-Time Java Implementation. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002* (Irvine, CA, USA, October 2002), R. Meersman and Z. Tari, Eds., Springer, pp. 900–921.

[55] Deters, M., and Cytron, R. K. Automated Discovery of Scoped Memory Regions for Real-Time Java. *SIGPLAN Notices 38*, 2 supplement (2003), 25–35.

[56] Deters, M., Leidenfrost, N., and Cytron, R. K. Translation of Java to Real-Time Java Using Aspects. In *Proceedings of the International Workshop on Aspect-Oriented Programming and Separation of Concerns* (Lancaster, UK, 2001). Proceedings published as Tech. Rep. CSEG/03/01 by the Computing Department, Lancaster University.

[57] Deutsch, A. On the Complexity of Escape Analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)* (Paris, France, January 1997), ACM Press, pp. 358–371.

[58] Domani, T., Kolodner, E. K., and Petrank, E. A Generational on-the-fly Garbage Collector for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)* (Vancouver, British Columbia, Canada, June 2000), ACM Press, pp. 274–284.

[59] Fabri, J. Automatic Storage Optimization. In *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction (SIGPLAN '79)* (Denver, CO, USA, 1979), ACM Press, pp. 83–91.

[60] Filman, R. E., Elrad, T., Clarke, S., and Akşit, M., Eds. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[61] Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. M. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)* (Kaiserslautern, Germany, 1993), vol. LNCS 707 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 406–431.

[62] Garbervetsky, D., Nakhli, C., Yovine, S., and Zorgati, H. Program instrumentation and run-time analysis of scoped memory in java. In *International Workshop on Runtime Verification (RV '04)* (Barcelona, Spain, March 2004), vol. 113 of *Electronic Notes in Theoretical Computer Science*, European Joint Conferences on Theory and Practice of Software, Elseiver, pp. 113–121.

[63] Garey, M. R., and Johnson, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[64] Gay, D., and Aiken, A. Memory Management with Explicit Regions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)* (Montreal, Quebec, Canada, June 1998), ACM Press, pp. 313–323.

[65] Gay, D., and Aiken, A. Language Support for Regions. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)* (Snowbird, UT, USA, June 2001), ACM Press, pp. 70–80.

[66] George O. Collins, J. Experience in Automatic Storage Allocation. *Communications of the ACM 4*, 10 (1961), 436–440.

[67] Gergov, J. Algorithms for Compile-Time Memory Optimization. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '99)* (Baltimore, MD, USA, January 1999), Society for Industrial and Applied Mathematics, pp. 907–908.

[68] Gosling, J., and Bollella, G. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[69] Gosling, J., Joy, B., Steele, G., and Bracha, G. *Java Language Specification, Second Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[70] Hanson, D. R. Fast Allocation and Deallocation of Memory Based on Object Lifetimes. *Software Practice and Experience 20*, 1 (1990), 5–12.

[71] Hardy, G. *An Introduction to the Theory of Numbers*, fifth ed. Oxford Science Publications. Oxford Press, 1979.

[72] Hertz, M., and Berger, E. Automatic vs. Explicit Memory Management: Settling the Performance Debate. Tech. Rep. CS TR-04-17, University of Massachussetts, Department of Computer Science, 2004.

[73] Higuera, T., Issarny, V., Banâtre, M., and Parain, F. Memory Management for Real-Time Java: An Efficient Solution using Hardware Support. *Real-Time Systems Journal 26*, 1 (2004), 63–87.

[74] Hirschberg, D. S. A Class of Dynamic Memory Allocation Algorithms. *Communications of the ACM 16*, 10 (1973), 615–618.

[75] Holte, R. C., and Hernádvölgyi, I. T. A Space-Time Tradeoff for Memory-Based Heuristics. In *Proceedings of the 16th National Conference on Artificial Intelligence and the 11th Conference on Innovative Applications of Artificial Intelligence (AAAI '99/IAAI '99)* (Orlando, FL, USA, 1999), American Association for Artificial Intelligence, pp. 704–709.

[76] International Organization for Standardization (ISO). *Information Technology – Portable Operating System Interface (POSIX) – ISO/IEC 9945* . Genéve, Switzerland, 2003.

[77] Johnstone, M. S. *Non-Compacting Memory Allocation and Real-Time Garbage Collection.* PhD thesis, The University of Texas at Austin, December 1997.

[78] Johnstone, M. S., and Wilson, P. R. The Memory Fragmentation Problem: Solved? *SIGPLAN Notices 34*, 3 (1999), 26–36.

[79] Joseph, M., and Pandya, P. Finding Response Times in a Real-Time System. *Computer Journal 29*, 2 (October 1986), 390–395.

[80] Kalyanasundaram, B., and Pruhs, K. Dynamic Spectrum Allocation: The Impotency of Duration Notification. In *Procedings of the 20th Conference on Foundations of Software Technology and Theoretical Computer Science, (FST TCS '00)* (New Delhi, India, December 2000), S. Kapoor and S. Prasad, Eds., no. LNCS 1974 in Lecture Notes in Computer Science, Springer, pp. 421–428.

[81] Kernighan, B. W. *The C Programming Language.* Prentice Hall Professional Technical Reference, 1988.

[82] Kierstead, H. A. A Polynomial Time Approximation Algorithm for Dynamic Storage Allocation. *Discrete Mathematics 88*, 2–3 (1991), 231–237.

[83] Knowlton, K. C. A Fast Storage Allocator. *Communications of the ACM 8*, 10 (1965), 623–624.

[84] Krishna, A. S., Schmidt, D. C., and Klefstad, R. Enhancing Real-Time CORBA via Real-Time Java Features. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)* (Tokyo, Japan, March 2004), IEEE Computer Society, pp. 66–73.

[85] Kwon, J., and Wellings, A. J. Memory Management Based on Method Invocation in RTSJ. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '04)* (Agia Napa, Cyprus, October 2004), R. Meersman, Z. Tari, and A. Corsaro, Eds., vol. LNCS 3292 of *Lecture Notes in Computer Science*, Springer, pp. 333–345.

[86] Kwon, J., Wellings, A. J., and King, S. Ravenscar-Java: a High-integrity Profile for Real-time Java. *Concurrency and Computation: Practice and Experience 17*, 5-6 (April 2005), 681–713.

[87] Lea, D. A Memory Allocator. Available at: http://gee.cs.oswego.edu/dl/html/malloc.html.

[88] Lindblad, J. Reducing Memory Fragmentation. *Embedded Systems Engineering* (April 2004), 26–28.

[89] Lindholm, T., and Yellin, F. *Java Virtual Machine Specification, Second Edition.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[90] Luby, M. G., Naor, J. S., and Orda, A. Tight Bounds for Dynamic Storage Allocation. *SIAM Journal on Discrete Mathematics 9*, 1 (1996), 155–166.

[91] Mann, T., Deters, M., LeGrand, R., and Cytron, R. K. Static Determination of Allocation Rates to Support Real-Time Garbage Collection. *SIGPLAN Notices 40*, 7 (2005), 193–202.

[92] McDowell, C. E. Reducing Garbage in Java. *SIGPLAN Notices 33*, 9 (1998), 84–86.

[93] Miranda, J., and Schonberg, E. GNAT: On the Road to Ada 2005. *Ada Letters XXIV*, 4 (2004), 51–60.

[94] Naor, J., Orda, A., and Petruschka, Y. Dynamic Storage Allocation with Known Durations. *Discrete Applied Mathematics 100*, 3 (March 2000), 203–213.

[95] Neely, M. S. An Analysis of the Effects of Memory Allocation Policy on Storage Fragmentation. Master's thesis, The University of Texas at Austin, May 1996.

[96] Niessner, A., and Benowitz, E. RTSJ Memory Areas and Their Affects on the Performance of a Flight-Like Attitude Control System. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '03)* (Catania, Sicily, Italy, November 2003), R. Meersman and Z. Tari, Eds., vol. LNCS 2889 of *Lecture Notes in Computer Science*, Springer, pp. 508–519.

[97] Nilsen, K. D., and Gao, H. The Real-Time Behavior of Dynamic Memory Management in C++. In *Proceedings of the 1st IEEE Real-Time Technology and Applications Symposium (RTAS '95)* (Chicago, IL, USA, May 1995), IEEE Computer Society, pp. 142–153.

[98] Palacz, K., Baker, J., Flack, C., Grothoff, C., Yamauchi, H., and Vitek, J. Engineering a Customizable Intermediate Representation. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME '03)* (San Diego, California, June 2003), ACM Press, pp. 67–76.

[99] Pizlo, F., Fox, J. M., Holmes, D., and Vitek, J. Real-Time Java Scoped Memory: Design Patterns and Semantics. In *7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)* (Vienna, Austria, May 2004), IEEE Computer Society, pp. 101–110.

[100] Puaut, I. Real-Time Performance of Dynamic Memory Allocation Algorithms. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS'02)* (Vienna, Austria, June 2002).

[101] R. Klefstad, D. C. S., and O'Ryan, C. The Design of a Real-time CORBA ORB using Real-time Java. In *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing* (April 2002).

[102] Robertz, S. G., and Henriksson, R. Time-Triggered Garbage Collection: Robust and Adaptive Real-Time GC Scheduling for Embedded Systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tool for Embedded Systems* (New York, NY, USA, 2003), ACM Press, pp. 93–102.

[103] Robson, J. M. An Estimate of the Store Size Necessary for Dynamic Storage Allocation. *Journal of the ACM 18*, 3 (1971), 416–423.

[104] Robson, J. M. Bounds for Some Functions Concerning Dynamic Storage Allocation. *Journal of the ACM 21*, 3 (1974), 491–499.

[105] Robson, J. M. Worst Case Fragmentation of First-fit and Best-fit Storage Allocation Strategies. *The Computer Journal 20*, 3 (1977), 242–244.

[106] Ross, D. T. The AED free storage package. *Communications of the ACM 10*, 8 (1967), 481–492.

[107] Schmidt, D. C., Rohnert, H., Stal, M., and Schultz, D. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* John Wiley & Sons, Inc., New York, NY, USA, 2000.

[108] Shaham, R., Kolodner, E. K., and Sagiv, M. Heap Profiling for Space-Efficient Java. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, UT, USA, 2001), ACM Press, pp. 104–113.

[109] Shannon, C. E., and Weaver, W. The Mathematical Theory of Communication. University of Illinois Press, 1949.

[110] Siebert, F. Eliminating External Fragmentation in a Non-Moving Garbage Collector for Java. In *Proceedings of the 2000 international conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '00)* (San Jose, CA, USA, November 2000), ACM Press, pp. 9–17.

[111] Slusarek, M. A Coloring Algorithm for Interval Graphs. In *Mathematical Foundations of Computer Science 1989 (MFCS'89)* (Porabka-Kozubnik, Poland, August 1989), A. Kreczmar and G. Mirkowska, Eds., vol. 379 of *Lecture Notes in Computer Science*, Springer, pp. 471–480.

[112] Stefano, A. D., Fargetta, M., Pappalardo, G., and Tramontana, E. Separating Soft Real-Time From Functionality Concerns in Complex Object Oriented-Applications. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '04)* (Agia Napa, Cyprus, October 2004), R. Meersman, Z. Tari, and A. Corsaro, Eds., vol. LNCS 3292 of *Lecture Notes in Computer Science*, Springer, pp. 406–417.

[113] Taft, S. T., Duff, R. A., Brukardt, R., and Plödereder, E., Eds. *Consolidated Ada Reference Manual. Language and Standard Libraries, International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1*, vol. 2219 of *Lecture Notes in Computer Science.* Springer, 2002.

[114] Tanenbaum, A. S. *Operating Systems: Design and Implementation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[115] The J-Consortium. International J-Consortium Specification – High Integrity Profile Revision 0.2. Available at: http://www.opengroup.org.

[116] Tofte, M., Birkedal, L., Elsman, M., and Hallenberg, N. A Retrospective on Region-Based Memory Management. *Higher-Order and Symbolic Computation 17*, 3 (2004), 245–265.

[117] Tofte, M., and Talpin, J.-P. Implementing the Call-by-Value Lambda-Calculus using a Stack of Regions. In *Proceedings of the 21st ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL '94)* (Portland, OR, USA, January 1994), ACM Press, pp. 188–201.

[118] Tofte, M., and Talpin, J.-P. Region-Based Memory Management. *Information and Computation 132*, 2 (1997), 109–176.

[119] Tsang, S. L. An Evaluation of AOP for Java-Based Real-Time Systems Development. Master's thesis, University of Dublin, September 2003.

[120] Vera, X., Lisper, B., and Xue, J. Data Caches in Multitasking Hard Real- Time Systems. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS '03)* (Cancun, Mexico, December 2003), IEEE Computer Society, pp. 154–165.

[121] Watkins, D., Hammond, M., and Abrams, B. *Programming in the .Net Environment.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[122] Wellings, A. J. Is Java Augmented with the RTSJ a Better Real-Time Systems Implementation Technology than Ada 95? In *Proceedings of the 12th International Real-Time Ada Workshop (IRTAW 12)* (2003), vol. XXIII of *Ada Letters*, pp. 16–21.

[123] Wellings, A. J., Clark, R., Jensen, D., and Wells, D. A Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation. In *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '02)* (April 2002), pp. 13–22.

[124] Williams, T., Ed. *Analysis and Design of Hierarchical Control Systems with Special Reference to Steel Plant Operations.* Advances in Industrial Engineering, 3. Elsevier, North-Holland, 1986.

[125] Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the International Workshop on Memory Management* (Kinross, Scotland, UK, 1995).

[126] Zhao, T., Noble, J., and Vitek, J. Scoped Types for Real-Time Java. In *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS '04)* (Lisbon, Portugal, December 2004), IEEE Computer Society, pp. 241–251.

[127] Zorn, B. Barrier Methods for Garbage Collection. Tech. Rep. CU-CS-494-90, University of Colorado at Boulder, 1990.