Reflexes: Programming Abstractions for Highly Responsive Computing in Java

THÈSE N^{O} 4228 (2008)

PRÉSENTÉE LE 31 OCTOBRE 2008

À LA FACULTE INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE PROGRAMMATION DISTRIBUÉE
PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Jesper Honig SPRING

acceptée sur proposition du jury:

Prof. M. Odersky, président du jury
Prof. R. Guerraoui, Prof. J. Vitek, directeurs de thèse
Prof. B. Garbinato, rapporteur
Prof. V. Kuncak, rapporteur
Prof. O. Lehrmann Madsen, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2008

Abstract

Achieving sub-millisecond response times in a managed language environment such as Java introduces significant implementation challenges. The Achilles' heel of Java is its reliance on automatic memory management for reclaiming dead objects. Typically, the garbage collectors used in commercial Java virtual machines are designed to maximize the performance of applications at the expense of predictability. Moreover, it is non-deterministic as to *when* and for *how long* the garbage collector will run. As a consequence garbage collection introduces execution interference that can easily reach hundreds of milliseconds, preventing the timeliness requirements of the real-time systems from being satisfied.

Another source of interference relates to the integration of real-time tasks with a time-oblivious application. Typical programming practices for sharing data between threads involve synchronized access to some shared data structure through mutual exclusion. In a real-time system this might lead to unbounded blocking of the real-time thread, so-called *priority inversion*, causing serious deadlines infringements.

Faced with these challenges a system designer typically has two options: to deploy a real-time garbage collector (RTGC), or to restrict the programming model. RTGCs achieve much improved predictability over traditional *stop-the-world* garbage collectors by interleaving the application execution with the garbage collection. Recent advances in real-time garbage collection algorithms have reduced latency to approximately 1 millisecond. Nevertheless, some applications, e.g., safety critical applications certified according to DO-178B, have temporal requirements beyond what is possible with state-of-the-art RTGCs. Moreover, real-time garbage collectors still face the problem of priority inversion.

This thesis presents *Reflexes*, a simple, statically type-safe restricted programming model facilitating the construction of highly-responsive applications in Java. Reflexes are designed to make it easy to write and integrate simple periodic tasks or complex stream processors, both observing real-time timing constraints in the sub-millisecond range, into larger time-oblivious Java applications. Reflex tasks run in a part of memory free from garbage collection interference, and maintain a class-based separation between object lifetimes, enabling reclamation of periodic garbage in constant time. Tasks are organized in a graph and communicate through uni-directional, non-blocking channels. Furthermore, Reflexes enable non-blocking interaction between real-time tasks and time-oblivious code using methods with transactional semantics, circumventing typical problems of *priority inversion* when using common programming practices for synchronizing access to shared data. Reflexes specify a set of static safety checks preventing dangling pointers and preventing a Reflex from observing heap-allocated objects in an inconsis-

tent state as they are, e.g., when being copied by a garbage collector. These checks are enforced statically by an extension of the standard Java compiler to guarantee correctness.

We describe two implementations of Reflexes: a stand-alone prototype implementation built on top of a research Java real-time virtual machine, and an implementation based on an integration of Reflexes with two existing restricted programming models into a single unified framework, *Flexotask*, running on top of an industrial-strength Java real-time virtual machine. Whereas the prototype implementation of Reflexes is limited to a virtual machine with uni-processor support and exploits non-standard features of the virtual machine, the latter implementation leverages the multi-processor support of an industrial-strength virtual machine with minimal extensions. For both implementations we report separately on a number of encouraging results from empirical experiments using benchmark and real-world applications. Specifically, our experiments show that Reflexes in both cases are capable of achieving sub-millisecond response time with a high degree of predictability.

Keywords: Real-time systems, Java virtual machine, Memory Management, Ownership types.

Résumé

Dans un langage de programmation interprété comme Java, où les programmes s'exécutent à l'aide d'une machine virtuelle, il est difficile d'atteindre des temps de réponses en dessous de la milliseconde. Le talon d'Achille du langage Java dans un contexte temps réel est sa dépendance au ramasse-miettes collectant les objets morts. Les ramasses-miettes implémentés dans les machines virtuelles commerciales sont conçus pour maximiser la performance au prix d'une baisse de leur prédictabilité. De plus, le moment où ils se déclenchent tout comme la durée de leur exécution sont non déterministes. L'exécution d'un ramasse-miettes interfère avec celle du programme, causant facilement des retards de plusieurs centaines de millisecondes, ce qui rend difficile le respect des contraintes temporelles du programme.

Une autre source d'interférence émerge lorsqu'une tâche temps réel se mêle à des tâches sans contrainte temporelle. Lorsque des tâches différentes accèdent à une même ressource, une abstraction d'exclusion mutuelle est utilisée pour réglementer l'ordre d'accès à cette ressource. Dans un système temps réel, cette abstraction peut s'avérer dangereuse dans la mesure où une tâche temps réel peut se retrouver bloquée à attendre une tâche de plus faible priorité. Ce phénomène est appelé *inversion de priorité* et peut causer de graves dépassement des délais impartis á la tâche temps réel.

Face à ces défis, un concepteur de systèmes a en général deux options: déployer un ramasse-miettes temps réel (RTGC pour Real-Time Garbage Collector) ou restreindre le modèle de programmation. Un ramasse-miettes temps réel atteint une bien meilleure prédictabilité que les ramasses-miettes traditionnels, basés sur une approche *pause-pour-maintenance* (stop-the-world), en intercalant leur exécution tout au long de l'exécution du programme même. Des avancées récentes dans ce domaine ont réduit la latence aux environs d'une milliseconde. Néanmoins, certaines applications, comme les applications critiques certifiées D0-178B, ont des contraites temporelles que l'on ne peut atteindre avec ce type de ramasse-miettes.

Cette thèse présente *Reflexes*, un modèle de programmation restreint à la fois simple, sûr au niveau du typage et qui facilite la conception d'applications Java hautement réactives demandant une prédictabilité en dessous de la milliseconde. Les Reflexes sont conçus de manière à faciliter l'écriture et l'intégration de tâches périodiques simples et de traitements complexes de flux dans des applications n'ayant pas de contraintes temporelles tout en respectant des délais impartis de moins d'une milliseconde. Pour ce faire, les tâches Reflex s'exécutent dans une partie mémoire où le ramasse-miette n'intervient pas et séparent en mémoire les objets de différentes classes, ce qui permet de récupérer l'espace alloué périodiquement par ces tâches en un temps constant. Ces tâches, communiquant par des canaux unidirectionnels non bloquants, forment un graphe.

De plus, Reflexes autorise l'interaction non bloquante entre des tâches temps réels et non temps réel en utilisant des sémantiques transactionnelles, ce qui évite le problème typique d'inversion de priorité. Le modèle de programmation Reflexes spécifie un ensemble de règles statiques de sécurité pour éviter des pointeurs pendants et empêche une tâche Reflex d'observer des objets, alloués sur le tas, dans un état inconsistent (par exemple s'ils sont en train d'être manipulés par le ramasse-miettes). Ces règles sont contrlées statiquement par une extension du compilateur Java afin de garantir l'exécution correcte du programme.

Ce document présente deux implémentations de Reflexes: un prototype autonome implémenté sur une machine virtuelle temps réel Java issue du monde de la recherche et une implémentation basée sur l'intégration de Reflexes dans deux modèles de programmation existant, sur la plateforme unifiée *Flexotask*. Cette dernière s'exécute sur une machine virtuelle temps réel industrielle Java de référence. Alors que le prototype de Reflexes est limité aux machines monoprocesseur et exploite des fonctions non standard de la machine virtuelle, la seconde implémentation tire parti des architectures multiprocesseurs en ajoutant un minimum d'extensions à la machine virtuelle. Pour les deux implémentations, cette thèse présente des résultats encourageant d'expérimentations, utilisant à la fois des jeux d'essai et de vraies applications. Plus particulièrement, les expériences montrent que Reflexes est capable dans les deux cas de respecter des délais en dessous de la milliseconde tout en assurant un très haut degré de prédictabilité.

Mots-clés: Systèmes temps réel, machine virtuelle Java, gestion mémoire, types de propriété.

Acknowledgements

During my four and a half years at EPFL, I have met an amazing number of bright, funny, grazy and exciting people from all over the world. Many of these have had a profound impact on either me or my work, and thus deserve a proper token of appreciation.

Before starting, however, I would like to take the opportunity to thank Professor Martin Odersky for presiding over my thesis exam, and thank the other five members of the jury for the time spent examining my thesis and participating in my private defense.

First and foremost, I would like to thank my advisor Professor Rachid Guerraoui for accepting me as a Ph.D. student in his LPD laboratory. Though my research path ended up in a different direction from that of the laboratory, I have been extremely grateful for Rachid's continuous interest and encouragements, for always pushing for results and keeping me to the fire. In this light, I am even more grateful for Rachid's unreluctant willingness to always fund my travels and other needs in order to achieve my research goals! Finally, thanks for forcing me to stay physically in shape through our countless jogging trips to Morges and back, for all our interesting discussions on *"interesting aspects of life"*, and for having had a lot of fun times together!

I am deeply in debt to Professor Jan Vitek of Purdue University for a collaboration that to me has been close to perfection, limited only by distance. Jan's calm attitude, steady focus and continuing pursuit of our goals has been truly motivating and inspiring! I am also very grateful that Jan in January 2008 accepted to be my co-thesis director, for always taking time to give advise, and for paying my way several times to visit his S3 laboratory at Purdue University. Also, thanks to Jan's wife and kids for making me feel welcome in their home on several occasions.

While at Purdue, I met a number of intesting people that contributed to my successful visits, among others Antonio Cunei, Filip Pizlo, Jean Privat, and Jaques Thomas. Special thanks goes to my good friend Patrick (Bob) Eugster for allowing my to stay at his place several times, showing me around West Lafayette, and of course for our interesting countless discussions on *"frustrating aspects of life"*.

From IBM Research, Hawthorne, New York, I would sincerely like to thank Joshua Auerbach and David F. Bacon for a very motivating, serious and fruitful collaboration that dates back to October 2007. Furthermore, I want to thank them and Erik Altman for hosting me during my 2 months internship there, which I really enjoyed professionally as well as privately. Special thanks goes to Joshua Auerbach, who not only stands out as one of the most brilliant computer scientists I have ever met, but also has been perfect and inspiring to work with (though I constantly struggled to keep up)!

From the EU-funded PalCom project, which I participated in during 4 years on behalf of EPFL, I would like to thank Peter Andersen, Lars Bak, Kasper Verdich Lund, and my friend Dominic Greenwood for a good collaboration. Thanks to Ole Lehrman Madsen for his efforts to team me up with nice-looking girls during my days as single, and of course for his unforgettable, nocturnal interpretation of *O´ solo mio* on the streets of Siena, Italy.

I am also thankful to all the other people I was brought to work with. Among these, I would like highlight Benoît Garbinato for his motivating support and time during the first year or two of my work, as well as all the current and past members of the LPD laboratory for the nice time I have spent there. Especially, I am grateful to my friends Sébastien Baehni, Ron Levy, Maxime Monod and Bastian Pochon for helping me settle in Lausanne and constantly assisting me with the french language. Our laboratory secretary, Kristine deserves a very big thank for always caring for us students, for all her efforts in providing for a pleasent social lab-life, and for simply being a very nice person!

Thanks to all my good friends in Denmark, especially Esben Krag Hansen and Per Qvist-Sørensen, and those friends living in Lausanne for all their encouragements, and for always being there for me.

Finally, I would like to thank the world's best parents, Benedikte Birgitte Honig Jensen and Jens Ove Spring for their endless love and support throughout my life and in particular through this swiss endeavour. Also, endless thanks to my sister Heidi, her family, and of course to the italian part of my family. Last but not least, thanks to my beautiful wife Gaetana for having the courage to share her life with a computer scientist, and for having worked over-time at home for the last 2 years to yield me time to work on my Ph.D. – *tu sei veramente la mia storia importante.*

Preface

This thesis describes the Ph.D. work I did at the School of Computer and Communication Sciences, EPFL, under the joint supervision of Professor Rachid Guerraoui, EPFL, and Professor Jan Vitek, Purdue University, during the period April 2004 to September 2008.

The thesis also includes and describes work that is the outcome of my collaboration with Joshua Auerbach and David F. Bacon of IBM Research, a collaboration that started in October 2007, and which included a two month internship at IBM Research, T.J. Watson Research Center, Hawthorne, New York, USA in March and April 2008.

The thesis focuses on programming abstractions for achieving highly responsive computing in Java and its content is based on the following published, peer-reviewed articles, or articles under submission.

[SGV08] *Integrating Hard Real-Time Tasks into Java with Reflexes.* Jesper H. Spring, Rachid Guerraoui, and Jan Vitek. To be submitted to *ACM Transactions on Embedded Computing Systems – Special Issue on Java Technologies for Real-Time and Embedded Systems (JTRES)*, 2008.

[ABG+08] *Flexible Task Graphs: A Unified Restricted Thread Programming Model for Java.* Joshua Auerbach, David F. Bacon, Rachid Guerraoui, Jesper H. Spring, and Jan Vitek. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2008.

[SPGV07a] *Reflexes: Abstractions for highly responsive systems.* Jesper H. Spring, Filip Pizlo, Rachid Guerraoui and Jan Vitek. In *Proceedings of the 3rd International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, 2007.

[SPGV07b] *Streamflex - High-throughput stream programming in Java.* Jesper H. Spring, Jean Privat, Rachid Guerraoui and Jan Vitek. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[GGH+07] *Pervasive Computing with Frugal Objects.* Benoit Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper H. Spring. In *Symposium on Pervasive Computing and Ad Hoc Communications (PCAC)*, 2007.

[GGH+06] *Frugal Mobile Objects.* Benoit Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper H. Spring. In *Proceedings of the Euro-American Workshop on Middleware for Sensor Networks, co-located with the 2nd International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2006.

[GGH+05] *Frugal Mobile Objects.* Benoit Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper H. Spring. Technical Report, 2005.

Finally, related to this work, as part of their Innovation Cluster for Embedded Software (ICES) initiative, Microsoft Research and Microsoft Switzerland announced on the 28th of August 2008 their willingness to sponsor funding of a 6 man year project within the LPD laboratory based on a grant application describing the results presented in this thesis as a starting point for an extension hereof and an implementation in C#, see press release [Micb].

Table of Contents

List of Figures

List of Tables

Part I

Introduction

1 Introduction

1.1 Motivation

This thesis is concerned with programming abstractions enabling the programming of highly responsive tasks in the Java programming language [GJSB00], and their integration and interaction with larger time-oblivious applications running on the same Java virtual machine.

Highly responsive systems are typically used for applications considered mission critical, such as embedded control systems for airplanes, financial applications, etc. With response times ranging from 10 microseconds to 10 milliseconds and the consequence of missed deadlines ranging between an annoyance and catastrophe, these systems must be carefully engineered.

What characterizes such real-time processing systems are their temporal constraints; they must react to stimuli from some external environment within a given time to avoid logical failure. Moreover, in contrast to traditional computing systems, the logical correctness of a real-time system is determined not only in terms of compliance with expected program behavior, but also in terms of timeliness of the behavior. Here, timeliness does not necessarily imply speed. In fact, it is a misconception that real-time systems are fast. Rather, the main difference between real-time and non-real-time systems is concerned with the constraints and predictability of the response times. In order to comply with temporal constraints, real-time systems are constructed to observe explicit timeliness requirements in their various response times.

Traditionally, real-time systems have been dedicated to solving a particular real-time processing task. However, the nature and role of real-time systems are evolving. Today, there is a trend of integrating small real-time tasks with strong temporal constraints into mainstream applications having no time constrained code. As examples hereof, both Sun Microsystems and BEA have

introduced real-time application servers. Consider also the example of the US Navy's DD-1000 Zumwalt class destroyer, which is rumored to have millions of lines of code in its shipboard computing system of which only small parts have real-time constraints. What these applications have in common is that they do not look like traditional real-time systems - they are large, often in the million of lines of code, and complex. Yet parts of them place stringent predictability requirements on the execution environment and language implementation. Moreover, the increase in complexity causes a proportional increase in the engineering and programming efforts to realize these systems.

Over the last few years, the Java programming language has become a viable platform for developing real-time applications. One reason for this has been in order to consolidate the development efforts on a single platform and to increase productivity. Indeed, high-level programming languages such as Java provide significant software engineering benefits over so-called low-level programming languages, such as C [KR80], for the construction of complex applications as they provide rich libraries of programmer-friendly abstractions that together with their execution environments hide and manage many low-level details, most notably memory management. Furthermore, developing an application and its real-time subsystems using the same language is beneficial. In particular, the complexity of integrating real-time tasks with the non-real-time tasks can significantly exploit the benefit of having interfaces between the two in the same language. Also, by consolidating the development efforts on a single platform one can benefit from using the same development environment, debuggers, etc. As a consequence, we argue for environments that seamlessly support mixed mode execution between non-, soft- and hard-real-time tasks. The goal of this thesis is to study how to design such an environment in a high-level programming language, such as Java.

1.2 Challenges

Using Java as the target platform introduces significant implementation challenges. In fact, it is not at all obvious that Java is suitable for applications with stringent timeliness requirements, even if these are required only for a small number of specific tasks, as it introduces a number of non-trivial challenges that have to be addressed.

1.2.1 Garbage Collection Interference

The Java virtual machine is the source of many potential interferences impacting the predictability of program execution. These interferences stem from maintenance of global data structures, just-in-time compilation and, most notably, memory management. While internal data structures can be optimized, and just-in-time compilation avoided through ahead-of-time compilers, the Achilles' heel of Java is its reliance on automatic memory management used for reclaiming dead objects. Identifying and deallocating such dead objects inevitably introduces some execution overhead. This overhead can consist of additional internal space requirements, additional time requirements, or some combination of the two. Typically, the garbage collectors used in

commercial Java virtual machines are designed to maximize the performance of applications at the expense of predictability. Moreover, it remains non-deterministic *when* and for *how long* the garbage collector will kick in. Consequently, garbage collection introduces execution interference that often obstructs the timeliness requirements of the real-time systems from being satisfied. In fact, garbage collection related interference can easily reach hundreds of milliseconds, plausibly causing the real-time system to miss its deadlines.

1.2.2 Synchronous Communication

The interaction between the time-oblivious code and the real-time subsystems might also cause for serious execution interferences. Typical programming practices for sharing data between threads involve synchronization on access to some shared data structure through mutual exclusion. In a real-time system this might lead to *priority inversion*, where a time-critical thread running with high priority is blocked from accessing shared data by a time-oblivious, lower priority thread holding the lock on the data. To complicate matters, the time-oblivious thread might itself be preempted by the garbage collector for an unbounded period, thereby significantly worsening the violations of the real-time subsystem's timeliness requirements.

1.2.3 System Calls

Finally, any system calls performed by the real-time subsystems make up a separate problem as, by their nature, they leave the virtual machine. Being outside the virtual machine such calls are no longer under its control, and are thus subject to external execution interferences, which make such system calls difficult to reason about their effect on predictability. In this thesis we will not consider system calls further, but rather conclude that dealing with system calls in an automated way is far from trivial and should be considered a research topic in its own right. Instead, we thus make the assumption that the programmer is knowledgeable and aware of the consequences on predictability of using such system calls in the time-critical code. Later in this thesis we will demonstrate an example of using such system calls to write to an I/O device while still achieving a high predictability precision.

1.3 Problem Statement

The goal of the research presented in this thesis is to non-intrusively extend an existing high-level language (namely Java) and its run-time environment to support reliable real-time programming. Moreover, we strive to support the development of real-time applications with response times in the sub-millisecond range and their integration and interaction with larger time-oblivious applications in the same mainstream execution environment, without restricting or modifying legacy code.

Pursuing this goal, the body of work presented in this thesis is mainly focused on addressing the following challenges.

1.3.1 Circumventing Garbage Collection Interference

Faced with the challenges imposed by garbage collection interference, a system designer is typically left with possible two strategies in order to reduce or circumvent it. The first is to deploy a real-time garbage collector (RTGC). RTGCs achieve much better predictability than traditional *stop-the-world* garbage collectors used in standard Java virtual machines by interleaving the application execution with the garbage collection. In other words, with RTGCs the garbage collection phases are divided into short-running bounded periods between which the application mutator gets to run. These bounded periods increase the predictability of the garbage collection, but comes at the expense of application utilization.

Even though RTGCs do achieve a significant improvement in predictability over traditional garbage collection algorithms, they fall short on some of the challenges. While recent advances in real-time garbage collection algorithms have enabled the reduction of the latency to approximately 1 millisecond [BCR03], there are applications with temporal requirements that go beyond what is currently possible with state-of-the-art RTGCs. Of course, real-time applications may have very different temporal constraints, but for certain types of highly responsive applications, including those that are subject of this research, latencies of this magnitude may not be usable. For instance, the category of real-time applications referred to as *safety critical*, such as those certified according to the DO-178B standard used in the aviation industry, often have very strict temporal requirements that cannot be guaranteed by current state-of-the-art real-time garbage collectors. For such applications with latency requirements in the tens of microseconds, any garbage collection delay in the millisecond area will cause a high-frequency task to miss its deadline with devastating consequences. Furthermore, RTGCs do not address, and thus are still vulnerable to, execution jitter stemming from priority inversion problems when synchronizing access to data shared between time-oblivious code and the real-time subsystems.

The second applicable strategy is to restrict the programming model in order to avoid interference from the garbage collector. However, restricting the programming model to improve predictability comes at the expense of expressive power, i.e., how much a legal program can do. An example of such a restricted programming construct is the `NoHeapRealtimeThread` found in the Real-Time Specification for Java (RTSJ) [GB00]. The `NoHeapRealtimeThread` executes in a memory region that is out of reach from the garbage collector, though the regular public heap is managed by the garbage collector as normal. However, from within the memory region, the `NoHeapRealtimeThread` is restricted from referencing any heap-allocated objects as doing so could cause memory related errors. A memory region is an allocation context that provides a pool of memory shared by threads executing within it. Specifically, RTSJ enables the `NoHeapRealtimeThread` to allocate within two types of memory regions; `ImmortalMemory` and `ScopedMemory`. In both of these regions, individual allocated objects cannot be deallocated. In the former, the allocations remain for the lifetime of the application, whereas in the latter all allocations are purged together in constant time as soon as all threads exit.

While the `NoHeapRealtimeThread` achieves the goal of avoiding interference from the garbage collector by executing in a separate memory scope, it too has deficiencies. An important problem relates to the fact that a `NoHeapRealtimeThread` is indeed a thread. Being a thread, it can do

anything – it cannot statically be restricted. Instead, the virtual machine must assure referential integrity at run-time by ensuring that the `NoHeapRealtimeThread` does not reference any heap-allocated objects. These run-time checks are expensive and have been documented to introduce a significant overhead [PV06]. Aiming for sub-millisecond execution intervals, avoiding such a run-time overhead is imperative.

Another problem with the `NoHeapRealtimeThread` relates to the scoped memory in which it executes. Programming with scoped memory entails a loss of compositionality due to sensitivity of *where* at run-time an object is allocated. This significantly complicates reasoning about program correctness as it is impossible to check all possible code paths in a program statically. As a consequence, a successfully compiled program might at some point during run-time throw an unanticipated runtime exception due to a memory integrity violation caused by the unexpected location of some object. Hence, code that has to run in scoped memory requires greater care to prevent errors than normal heap-allocated code.

Furthermore, relating to interaction, the `NoHeapRealtimeThread` also suffers from an isolation problem. As such, a `NoHeapRealtimeThread` can block on a lock held by an ordinary Java thread, again an example of priority inversion leading to serious deadline misses. This can occur if both the `NoHeapRealtimeThread` as well as the ordinary Java thread have synchronized access to some shared data, e.g, some object allocated in `ImmortalMemory`.

- **Question:** *How can the latency introduced by garbage collection be circumvented such that sub-millisecond predictability is not compromised, yet avoiding the deficiencies of the* `NoHeapRealtimeThread`*?*

1.3.2 Ensuring Type-Safety

Circumventing garbage collection by using region-based memory is attractive, but unfortunately not without problems concerning how to ensure integrity of the memory region. Code executing in a memory region might reference objects outside its region introducing a number of potential memory related problems, such as the risk of having dangling pointers, or observing objects in an inconsistent state as they are, for example, when being copied by a garbage collector.

With this in mind, it is imperative to prevent such unsafe references from occurring. In the case of `NoHeapRealtimeThread`, unable to detect such unsafe accesses statically, the virtual machine must insert non-trivial read and write barriers around all reference operations to check for potential violations. Any violations detected will lead to dynamic exceptions, such as `IllegalAssignmentError` and `MemoryAccessError`, being thrown back to the `NoHeapRealtimeThread`. Thus, while these dynamic checks prevent unsafe references from occurring, the cost of doing so is significant, as described earlier. Moreover, the lack of static checking also makes application development error-prone and brittle as any such memory integrity errors are deferred until run-time.

- *How can the run-time checks maintaining the memory region integrity be avoided without compromising type-safety?*

1.3.3 Enabling Type-Safe, Non-Blocking Communication

Integrating time-oblivious code with time-critical tasks introduces challenges when it comes to the interaction between these. For instance, any synchronous interaction might cause priority inversion situations, which will lead to compromised predictability. This could happen if a running high priority real-time thread would have to block and wait for a lower priority ordinary, time-oblivious thread to complete some work.

As described above, ensuring the integrity of a memory region is imperative when executing code in a separate memory region. Interaction between time-oblivious code and real-time tasks becomes even more problematic when it comes to communicating with references to objects allocated in different memory contexts. In fact, running in a memory region, any reference passed between a real-time and an ordinary, time-oblivious thread might cause type safety to be compromised.

Furthermore, integrating real-time tasks with existing time-oblivious applications and having them interacting is only a viable option if such scheme can be realized without requiring extensive modification of the legacy code.

- *How can threads observing sub-millisecond temporal requirements interact with time-oblivious threads in a type-safe manner without sacrificing predictability, and without requiring extensive modification of legacy code?*

1.4 Approach

The problems we have described cross the boundaries between programming abstractions, programming languages and run-time system implementation. The thesis presents a multi-layered solution based on the following principles.

Real-time threads should never experience any interference from the garbage collector causing for a negative impact on predictability. To guarantee this, real-time threads should be able to take execution priority over, and preempt all other, threads running in the virtual machine, even the garbage collector. Only by granting such privileges to the real-time thread can memory management induced deadline infringements be circumvented.

Preempting the garbage collector introduces problems with respect to type safety since real-time threads might observe heap-allocated objects in an inconsistent state. This can occur if a copying garbage collector is preempted by the real-time task while in the process of moving some object. Therefore, real-time threads should be shielded from the garbage collectable area (the heap), and should in general be restricted from observing any other heap-allocated objects.

Real-time threads should be able to interact with ordinary, time-oblivious threads. However, this should be done strictly without impacting predictability. Any synchronous interaction between tasks must be avoided and any interaction must be subject to the same type restrictions as mentioned above. Only thereby can type-safety be guaranteed.

Finally, enforcing these type restrictions should be achieved such that they do not impose any negative performance impact on the real-time threads, nor prevent reuse of legacy code and standard libraries.

To address these challenges, we pursue an approach combining:

- **Priority Preemptive Thread Scheduling** – We rely upon a priority preemptive thread scheduling scheme enabling differentiation based on priority numbers between tasks running in the system. With this scheme high-priority real-time threads can preempt all other threads, even the garbage collector, thereby avoiding the normal garbage collection latency which is vital to achieving predictability at high frequencies.
- **Private Memory Regions** – We consider executing real-time threads within separate private memory regions, thereby enabling allocation of objects that are outside the reach of the garbage collector. However, memory regions do not eliminate the need for garbage collection of objects outside the regions, i.e., on the public heap, nor the latencies introduced when this process takes place.
- **Static Safety Checks** – Circumventing and preempting garbage collection through memory regions and priority preemptive scheduling is only safe in so far that, roughly speaking, there are no object references pointing in and out of the regions. We enforce the safety of region-based memory operations, yet permit certain operations such as those enabling interaction between threads, by considering a set of static safety checks as a conservative extension of the standard Java type system. These static safety checks also make it possible to bypass the expensive run-time checks performed by the virtual machine, helping to achieve a high degree of predictability.
- **Run-time Environment Support** – Achieving sub-millisecond response times is impossible without support from the underlying run-time environment. We rely on the Java virtual machine to support the necessary functionality, such as, region-based allocation, pinning of objects, software transactional memory and priority preemptive scheduling, enabling the execution of highly responsive applications. Although many of these features are found in an RTSJ-compliant virtual machine, we do not mandate such a virtual machine, but note that it eases the implementation efforts.

The outcome of this approach is a simple, restricted programming model, called *Reflexes*[1], facilitating the development of highly responsive computing applications in Java.

1.5 Limitations

For the purposes of this thesis we restrict our focus to the Java programming language [GJSB00] on a Java virtual machine [LY99]. Java is a pragmatic choice as it has become a mainstream

[1]Throughout this thesis, the name *Reflexes* is used to refer to both *Reflexes* [SPGV07a] and its superset *StreamFlex* [SPGV07b]. The motivation for using the term *Reflexes* instead of *StreamFlex* is to highlight that our approach is general, and thus not tied to stream processing particularly.

programming language with a high level of adoption due to its rich libraries and powerful tool support. However, we believe that the results presented in this thesis are general enough to be applied, directly or with modest modifications, to other high-level programming languages, such as C# [HWG03] on the .NET Common Language run-time [Mica].

1.6 Contributions

The contributions presented in this thesis include the following:

- **Programming Model** – We present the design of a simple, type-safe restricted programming model, *Reflexes*, facilitating the construction of highly responsive applications in Java. The Reflex programming model makes it easy write and integrate simple periodic tasks or complex stream processors, both observing real-time timing constraints in the sub-millisecond range, into larger time-oblivious Java applications.
- **Static Safety Checks** – To avoid the need to apply expensive checks during run-time, we describe an informal specification of a set of static safety checks inspired by ownership types for statically ensuring the safety of memory operations within a Reflex task while at the same time permitting communication with time-oblivious code. In particular, the static safety checks propose a novel notion of *implicit* ownership, rendering superfluous the need to declare ownership parameters on class declarations. Furthermore, the static safety checks are non-intrusive by permitting unmodified reuse of legacy code with no requirement to rewrite standard libraries.
- **Obstruction-free, Transactional Communication Scheme** – We describe a scheme to facilitate non-blocking communication between time-critical tasks and time-oblivious Java code based on obstruction-free, transactional communication, ensuring that the time-critical Reflex task will not violate its temporal requirements following interaction with time-oblivious code. Furthermore, we present a general design of the communication scheme enabling implementations on multi-processors, where it cannot be assumed that the release of the time-critical task causes for the immediate execution halt of a time-oblivious thread running in parallel.
- **Implementation and Integration** – We report on two implementations of the Reflex programming model. To demonstrate viability of the approach, we present a stand-alone prototype implementation of Reflexes on a research virtual machine which has a uni-processor design, and native support for transactional methods. In addition, we implement extensions to the `javac` compiler to support the static safety checks.

  To provide a more powerful and flexible programming model, we present an integration of the Reflex programming model with two existing restricted programming models from IBM Research into a unified programming model, *Flexotask*. To demonstrate the strength and flexibility of our approach, we also detail an implementation of Flexotask on a commercial virtual machine with a multi-processor design. Furthermore, we provide development tool support for Flexotask into the Eclipse IDE, among other features the static safety checking.

- **Empirical Evaluation** – For both our implementations, we report on a number of empirical evaluations of the ability of the programming model to enable the development of real-time applications providing (a) sub-millisecond response times with a high degree of precision, and (b) throughput better or comparable to equivalent application variants built for alternative approaches. In both cases we demonstrate encouraging results through benchmark applications running simple periodic tasks in isolation to demonstrate the baseline performance, as well as concurrently with communicating, ordinary Java threads to demonstrate predictability and performance when interacting.

1.7 Thesis Outline

The thesis is organized as follows:

Part I: Introduction

Chapter 1: Introduction motivated the thesis, highlighted the challenges that have to be addressed to achieve the research goals. Next, the problem statement of the thesis was presented, after which the approach pursued to reach the goal was described. Finally, the contributions of the thesis were presented.

Chapter 2: Related Work provides an overview of the target platform for our research, Java, and pinpoints the main challenges in achieving high predictability. The chapter then goes on to describe alternative approaches for improving predictability, ranging from specific techniques to programming models for real-time systems. Finally, different programming models that all restrict the expressive power for predictability are presented.

Part II: Programming Model

Chapter 3: The Reflex Programming Model provides an introduction to the *Reflex* programming model, its basic concepts, and demonstrates by example how to program a highly responsive real-time task, integrate it with a time-oblivious application, and have them communicate.

Chapter 4: Static Safety Checking provides an informal description of the set of static safety checks applied to enforce restrictions on the time-critical Reflex tasks in order to ensure type-safety and reasons about the correctness of the checks.

Part III: Implementation

Chapter 5: Reflex Implementation describes the prototype implementation of the Reflex programming model on a research real-time Java virtual machine designed with a uniprocessor design.

Chapter 6: Empirical Prototype Evaluation describes a number of empirical experiments of the Reflex prototype implementation on top of the research real-time Java virtual machine using benchmark and real-world applications, and presents some encouraging results.

Part IV: Integration

Chapter 7: Flexotask Integration describes the integration of the Reflex programming model together with Eventrons and Exotasks from IBM Research into a unified restricted programming model, *Flexotask*, and highlights differences in model and static safety checks following this integration.

Chapter 8: Flexotask Implementation highlights the most interesting implementation challenges involved in getting Reflexes to run (as part of Flexotask) on an industrial-strength real-time Java virtual machine designed with a multi-processor design.

Chapter 9: Empirical Evaluation describes a set of empirical experiments of the Flexotask implementation on top of the IBM WebSphere Real-Time VM using benchmark and real-world applications, and presents results clearly establishing that Flexotasks can achieve high predictability in a challenging workload environment.

Part V: Conclusion

Chapter 10: Conclusion provides a summary of the findings of this thesis, the contributions, and highlights some open problems that are avenues of future work.

2
Related Work

Having described the challenges faced when striving for highly responsive computing in a managed language environment, this chapter gives some background and surveys some of the most important alternative approaches, and compares them to approach adopted by the Reflex programming model presented in this thesis.

The structure of the chapter is as follows. We first give a brief overview of the relevant aspects of Java, the target platform for Reflexes, with emphasis on the fundamental reasons why managed languages introduce latencies affecting predictability. In other words, we will not provide a general introduction to Java, but rather assume that the reader is familiar with the language. Next, we introduce the general concept of real-time garbage collectors and describe how deploying a real-time garbage collector in general might increase predictability, yet for some types of applications still might not be sufficient to meet their temporal requirements. Thereafter, we describe programming models designed specifically for real-time computing, including some of their forerunner models, and point out their advantages when programming real-time systems. Finally, we describe various efforts to increase predictability and lower latencies even further than what is possible with standard real-time programming models through an approach based on restricting the expressiveness.

2.1 The Java Programming Language

Over the last decade, the Java programming language has become one of the most widely used programming languages in academia as well as in the industry. Java [GJSB00] was developed by Sun Microsystems in the mid nineties as a strongly typed, object-oriented language that derived much of its syntax from that of C; this familiarity is typically attributed as being one of the

major reasons for its rapid adoption. Although its inspiration came from C/C++, Java adopted a much simpler object model and dispensed many of the low-level facilities of C/C++ that were common sources of program errors, most notably facilities related to pointer arithmetic and memory management.

In order to satisfy one of the design goals of Java, that of being *platform independent* enabling the execution of the same unmodified program on several hardware architectures, Java programs are compiled into an intermediate representation – a simplified set of machine instructions specific to the Java platform. Through this instruction set, a Java program can be made to execute on any computing architecture with the help of a *virtual machine*, a platform dependent program written in native code that can interpret and execute the Java bytecode instructions on the specific architecture on which it runs. Multiple operating systems (the most ubiquitous: Microsoft Windows, Linux, Mac OS X, Sun Solaris) running on different computer architectures (Intel x86, x64, SPARC etc.) are currently supported by the Java virtual machine.

In the first versions of Java, the virtual machine ran in interpreted mode only, with significant performance deficiencies as a consequence – compared to an equivalent natively compiled program. In later versions followed *Just-In-Time* (JIT) compilers, which identifies most frequently used bytecodes, and compile these code *hot spots* into faster executing native code. The result of these JIT compilers are getting closer and closer to native performance.

Another feature that undoubtably helped to spark the rapid adoption of Java was its use of automatic memory management. Although automatic memory management by far was not something new, dating back to the sixties when applied to the Lisp programming language [MAE+84], Java successfully brought it to a mainstream language. In programming languages with manual memory management, the programmer is responsible for deallocating objects and return the now free space that they occupied back to the system. Failing to do so will result in *memory leaks* that eventually will exhaust the available memory and cause the entire program to fail. With automatic memory management, the programmer no longer has these concerns, or rather the concerns are much smaller. In fact, even with automatic memory management, some forms of memory leaks can still occur, e.g., if an application allocates a large number of temporary objects (or a few objects that take up large amounts of memory) that are referenced although they are no longer needed. In Java, an object is ready to be deallocated and its memory can be given back to the system only once it is not referenced from any other object. However, in general, when using automatic memory management, memory leaks are significantly less likely to occur than without. Central to the automatic memory management of Java is the garbage collector – built in to the virtual machine – that transparent to the programmer scavenges the memory (heap) at certain times for dead objects no longer referenced by any part of the running applications and reclaim the memory held by these. In standard Java virtual machines, this scavenging is performed in a *stop-the-world* manner, where all running user-threads in the virtual machine are paused during this process, introducing latencies in the hundreds of milliseconds. Combined with the fact that it remains non-deterministic *when* and for *how long* the garbage collector will step in to perform its duties, standard Java garbage collection is not suitable for the requirements of real-time applications.

Finally, in addition to the language, a virtual machine and a compiler, official Java implemen-

tations come with a rich set of standard class libraries, embracing high degree of development productivity and code reuse. Furthermore, a huge community surrounding Java provides numerous commercial and free development tools, including powerful debuggers, profilers and IDEs.

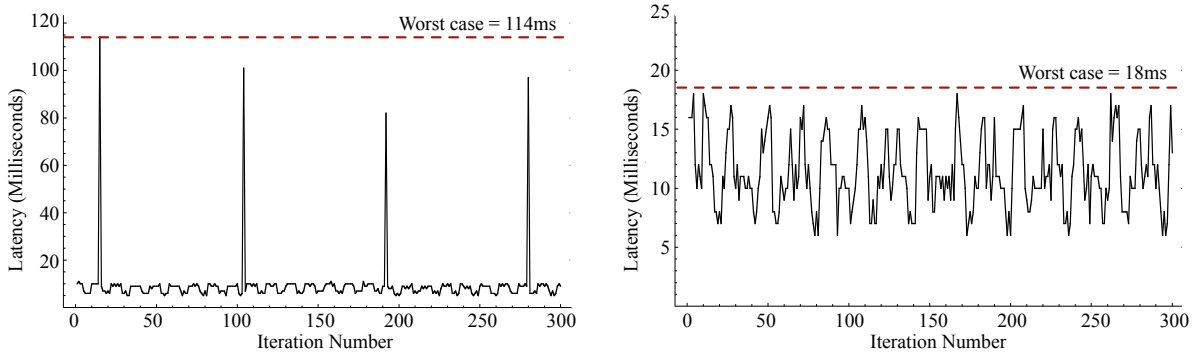
2.2 Real-Time Garbage Collectors

As pointed out, the standard Java garbage collection introduces latencies that not deterministically bounded. An alternative approach to normal garbage collection is to use real-time garbage collection algorithms, ensuring deterministic upper bounds on the latency. Using a real-time garbage collector is attractive as the semantics of the plain Java programs are left completely unaffected. Instead, using a real-time garbage collector simply just concerns the internals of the virtual machine.

However, there are some major drawbacks to the use of RTGC. For garbage collection to be used in hard real-time applications, non-trivial, reasonable estimates of worst-case bounds must be provided for: (a) garbage collection latency time, (b) throughput, and (c) allocation rate. Because events that arrive while the collector is operating cannot be handled until the collector yields, latency must be bounded. Furthermore, the mutator (i.e. application code) must not be interrupted too frequently, causing a drop in utilization, i.e., throughput. In particular, the effect on throughput must not be so severe as to prevent the application from handling events in a timely fashion. Finally, all memory allocation requests must succeed, and thus if the application has a high allocation rate, the garbage collector's time quantas must be correspondingly large, which inevitably comes at the expense of application utilization and thus throughput.

Work on real-time collection can be traced back to Bakers incremental copying collector [HGB78]. The central idea behind Bakers work is decreasing the intrusiveness of a collector by piggybacking work onto mutator operations. To ensure consistency, a small piece of code, called a read barrier, is inserted by the compiler before every memory read to perform copying, and the allocation code is modified to perform a bounded amount of collection work. The worst-case in a program using Bakers collector involves a copy operation upon every read, and a (large) unit of collection work on every allocation. Hence, even though individual pauses are small, the worst case execution time of an allocation makes Bakers collector unsuitable for usage in hard real-time systems. Put in another way, the garbage collection fails to bound its impact on throughput. Bakers collector is said to be work-based, in the sense that work done by the mutator leads to work by the collector.

As an alternative to work-based real-time collection, various time-based systems have been proposed [BCR03, Hen98, Det04]. In time-based systems, the principal is that the collector interleaves execution with the mutator at regular intervals. Constant-time read- (or write-) barriers are still needed to maintain consistency, but allocation can be made in constant time. The worst-case bounds on execution time in the mutator become more realistic, allowing the collector to be used in hard real-time systems. In fact, recent advances in real-time garbage collection for Java [BCR03] has made Java a viable platform for writing real-time applications whose latency requirements are in the millisecond range.



(a) Standard GC (b) RTGC

Figure 2.1: Benchmark application execution illustrating difference in garbage collection latency when running using (a) a standard copying garbage collector, and (b) a time-based real-time garbage collector. The x-axes show the iteration number and the y-axes the latency in milliseconds.

Fig. 2.1 illustrates the execution time of periodic processing in a benchmark application when executing on a Java virtual machine with (a) a standard copying garbage collector, and (b) a time-based real-time garbage collector. With the standard garbage collector, the mutator thread performs around 8 milliseconds of useful work in each iteration, but the garbage collector causes some iterations to take up to 114 milliseconds causing multiple deadline misses. With the real-time collector, the worst-case observed time is 18 milliseconds. This is interesting because, even if the bound on any individual pause is 1 millisecond, the mutator thread takes twice as long to complete because it is interrupted multiple times.

This clearly shows that pause times are only part of the cost of RTGC, one has to account for the overhead of barriers and the frequency of pauses. In fact, the real-time garbage collector approach currently appears limited to some lower bound on achievable latencies due to caching and context-switching effects. Though this lower bound is somewhere around the 1 millisecond range, there are still real-time applications having lower latency requirements than what can be achieved with state-of-the-art garbage collection algorithms. For these applications, latencies must be limited to the hundreds of microseconds.

2.3 Real-Time Programming Models

Even though a real-time garbage collector can significantly improve the predictability of an application, doing real-time programming in a general purpose programming language like Java is still cumbersome for the programmer. More specifically, being general purpose, such programming languages do not offer any language support for common real-time programming idioms enabling the programmer to control the temporal behavior of the program. This need has given rise to a number of real-time programming models.

Before the days of programming languages specifically designed for real-time computing, real-

time systems were built using a combination of standard languages in combination with some real-time operating system. With this approach, however, real-time properties like scheduling and communication were managed by the operating system, and thus to a large extent out of control of the programmer. Furthermore, the temporal aspects of the applications must be designed carefully with the underlying platforms in mind, adjusting various periods and worst case execution times accordingly. Inevitably, this dependency on the execution platform significantly limits the portability of such applications [Car02]. In contrast, programming languages exposing real-time related primitives increase the level of portability and reusability in addition to increasing general user-level control of the system behavior.

2.3.1 Concurrent Programming Models

The precursor to real-time programming languages as we know them today stem from the eighties with the advent of concurrent programming languages, like Concurrent ML [Rep93], Erlang [Arm97] and MultiLisp [RHH85]. What characterizes these concurrent programming languages is that they provide the programmer with explicit control over concurrency in programming model through some set of primitives enabling the creation of concurrent activities, and defining their possible interaction.

Generally speaking, these concurrent programming languages adopt one of two approaches to concurrency; synchronous or asynchronous. With the asynchronous model, system components can execute in parallel, independently and at different frequencies. In contrast, the synchronous model assumes the presence of some global clock that keeps all system component executions in synchrony.

Erlang, developed as base for the creation of large, dynamic software systems, such as telephone switches, is an example of a language exploiting the asynchronous model, specifically based on an actor-based [Agh86] approach. Generally, in an asynchronous model, activities are programmed to react independently to external stimuli, triggering their execution. Activities typically interact by some means of asynchronous communication, such as Futures [RHH85] (or Promises [LS88]) used in MultiLisp, shared memory abstractions, or message-passing systems as used by Erlang where messages are put on the activities' inbound queues (if they match some interest pattern) thereby triggering the subsequent execution of the receiver activities.

Although concurrent programming models effectively address the issue of handling concurrency, they are typically non-deterministic in their execution, i.e., the sequence with which the activities are executed is unknown at compile-time. Furthermore, the time-dependent handling is most often not explicit in the programming model, and their semantics often vague. Consequently, when it comes to classical concurrent systems it is non-trivial to reason about their predictability, a crucial property when temporal constraints of a real-time systems have to be satisfied.

2.3.2 Synchronous Programming Models

Synchronous languages, like Esterel [BG92], Lustre [CPHP87] and Signal [GGB87], build upon a deterministic concurrency model. Shared by this group of languages is a *perfect synchrony hypothesis* [BB02] that satisfies the temporal constraint through a conceptual assumption of instantaneousness within the system. Here, instantaneous means that reaction, computation and communication within the system takes no time, or, conceptually, that the program executes on an infinitely fast machine. While the assumption of infinitely fast machines is usable when verifying the correctness of synchronous programs, it must be relaxed to be implemented on real computers. Typically, a notion of logical time, with a global clock having predefined intervals, is used.

Hence, a synchronous program describes a total ordered sequence of activities that are executed at different logical ticks in a global clock. Activities occurring at the same clock tick are considered concurrent, and the remaining activities simply follow their given order according to the clock. Once triggered by an incoming event, activities perform computation and possibly communicate with other activities by generating new events in zero time. Consequently, activities are conceptually able to produce their outputs synchronously with their inputs within the same clock tick.

With this scheme determinism in the program execution is ensured, no matter how any concurrent activities are interleaved. One interesting question is, of course, how the *zero time* hypothesis is achieved in practice. Providing specialized hardware support is one possible approach [Ber91]. For instance, realizing that *zero time* is logically equivalent to *atomic* execution means that this condition can be satisfied by *synchronous hardware*. Moreover, it turns out that satisfying the perfect synchronous hypothesis of zero duration is not strictly necessary in order for a synchronous program to run correctly. Rather, the logical correctness criteria of synchronous programs is that any input can be processed, and possible results being output, before any new input can occur. For synchronous programs running on normal hardware architectures, this condition can be satisfied simply by increasing the time unit of the global clock.

Evidently, synchronous activities are much more predictable than their asynchronous counterparts in that (1) the ordering of activities is deterministic and well-known at development-time, (2) their execution times are known (conceptually assumed to be zero), and (3) the tick intervals of the global clock are known. Indeed, synchronous programming languages have some desired properties of real-time systems, such as a high degree of predictability. Nevertheless, the synchronous languages fall short on a number of counts for building real-time systems. For instance, with synchronous languages the notion of time relates to ordering of events rather than physical time. Furthermore, although several clocks can be applied in a system, for complex applications being strictly bound to the time ticks of the global clock might be rather limiting, for instance, when interacting directly with hardware.

2.3.3 Programming Models with Direct Real-Time Support

Several programming models providing real-time support directly have been proposed, the following standing out as the most recent variants; Ada95 [ANS] and Real-Time Java [GB00] (RTSJ). Furthermore, existing, widely-used languages, such as C [KR80] and C++ [Str91], have also been adopted for real-time computing by extending them with standardized interfaces to real-time primitives in the operating system, like the IEEE POSIX standard [CIEE94a] concerning real-time extensions to POSIX P1003.1 standard [CIEE94b].

Compared to the previous approaches of concurrent and synchronous programming languages, both RTSJ and Ada95 (through its Real-time Annex) provide direct support for real-time idioms (periodic and event-driven tasks) in the languages, making it easier for the programmer to obtain the desired real-time behavior in the application. Furthermore, having support directly in the language significantly increases the level of portability and reusability of the applications. Finally, since these programming languages in addition all are suitable for general purpose, they support a wider range of application types, with respect to temporal constraints, than that of previous approaches.

Both being object-oriented programming languages, Ada and Java provide significant productivity benefits to the programmer, particularly for programs with code-bases in the millions of lines. Through object-oriented programming features, such as inheritance, encapsulation, namespace management, the programmer can much better structure the project and exploit extensive reuse of code etc. Contrary, as an imperative language C does not share these programming benefits. Rather, the major advantage of C is in its support low-level facilities, such as giving direct access to memory addresses etc.. Whereas in Java such low-level facilities mostly are abstracted away or removed completely, Ada does provide some support through its System Programming Annex. C++ falls somewhere in between; being object-oriented, it too benefits from the above mentioned features. At the same time, building on C, C++ too provides rich support for low-level facilities. However, typically only small portions of an entire program really depend on this low-level support.

Concurrency support is crucial for most real-time systems, enabling various parts of the system to respond concurrently to different events (or periods) with different priorities. Ada95 uses a concurrency model based on a high-level abstraction, a task, rather than the lower-level thread model adopted by Java. The C language does not provide any concurrency model directly. Instead, with C/C++, the programmer is left few options but to explore external APIs, such as the IEEE POSIX thread extension standard [CIEE95], for concurrency. Although such APIs might provide the needed concurrency, given the fact that the language itself has no built in concurrency model, the programmer is forced to consider how different libraries affect the behavior of the application in a multi-threaded setup. Also, the POSIX specification enables flexibility in the implementation rendering the possibility of portability compromises. These considerations are not required for real-time programming languages with an integrated concurrency model.

It is well-known that the C/C++ programs can be unsafe, most notably relating to its memory management and pointer arithmetic. Though many of the errors relating to these can be eliminated through various analysis techniques, higher-level programming languages typically offer

better guarantees to avoiding such errors from ever happening. The type checking performed by the C/C++ compiler is typically rather liberal, leaving behind many potential errors in the code. In comparison, to achieve higher soundness, both the Ada95 and Java programming models promote stronger typing, and thus the type checking performed by the Ada and Java compilers inevitably will detect many more potential problems in the code. In fact, Ada goes a step further than that of Java, by for instance having strongly typed scalars.

In terms of memory management, though the language semantics does nothing to prevent it, Ada implementations typically do not have any garbage collection facility, unlike required by Java. As a consequence, a running Ada program is not subject to interference from a garbage collector, a key factor affecting predictability in Java. However, Ada puts the responsibility of doing storage reclamation on the programmer, a traditional disadvantage with C/C++ programs, attributing many program errors, such as memory leaks. In contrast, most RTSJ-implementations come with a the virtual machine providing support for a real-time garbage collector enabling low latencies, thereby providing a predictable run-time environment and abstractions for controlling the real-time aspects of the application.

2.4 Restricted Programming Models

Although real-time garbage collection can significantly reduce latency and increase predictability, there are applications that have temporal requirements beyond that. Pushing application latency requirements beyond what is achievable using real-time garbage collection algorithms, system designers are therefore faced with a dilemma, having to choose between adopting a new programming language providing better timing guarantees, or achieving the application timing requirements by restricting their preferred real-time programming language.

An interesting question here is, why one would restrict the programming language, especially given the inevitable loss of expressive power. Restricting a programming model to a subset is a pragmatic approach that serves two purposes; eliminating those parts in the programming model introducing complex (and perhaps ambiguous) semantics and high overheads to have a more reliable and temporally predictability programming environment, while concurrently exploiting that the remaining parts are now more specialized (given limited usage patterns in the restricted code), whereby better techniques can be applied to optimize those parts. The end result is a programming model subset promoting an efficient and predictable computational model.

In the Java community, the desire to achieve sub-millisecond latency has given rise to a number of restricted programming models, each making different trade-offs and emphasize different advantage. These include the `NoHeapRealtimeThread` construct of RTSJ, Eventrons and Exotasks from IBM Research, together with Reflex [SPGV07a] and StreamFlex [SPGV07b] presented in this thesis. Fig. 2.2 tries to illustrate how these different restricted programming models compare to each other in terms of their tradeoff of expressiveness and latency guarantees. Furthermore, the figure also illustrates how standard Java and Java with a real-time garbage collector compares.

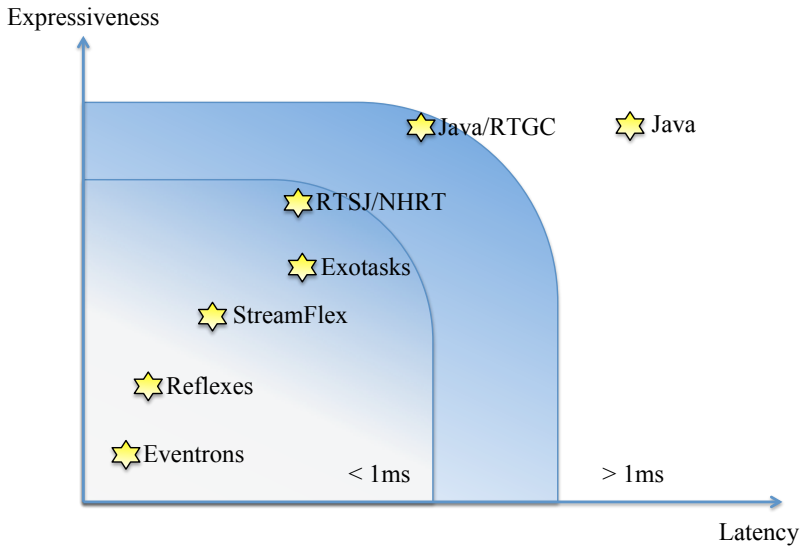


Figure 2.2: Comparing expressiveness versus worst case latencies of different restricted programming models for real-time Java. There is a tradeoff between latency guarantees and expressiveness. The RTSJ/NHRT is arguably the most expressive programming model with submillisecond latency, but it incurs throughput overheads due to run-time scope checks and faces the possibility of run-time failures. Contrary, Exotasks, StreamFlex, Reflexes, and Eventrons all rely on static checking and thus does not require the virtual machine to perform expensive run-time checks.

2.4.1 Real-time Specification for Java

RTSJ was developed within the Java Community Process as the first Java Specification Request (JSR-1) [Jav]. The RTSJ provides the means for applications to operate without interference from the garbage collector, and even preempt the garbage collector, while the collector thread is running.

Perhaps the most controversial feature added to RTSJ is the extension of the Java memory management model to include dynamically checked region-based memory. A memory region is an allocation context that provides a bounded pool of memory shared by threads executing within it. Individual objects allocated in a memory region cannot be deallocated; instead, all allocations made within the region are purged automatically in constant time as soon as all threads exit it.

A memory region is typically used to contain objects with roughly equivalent lifetimes. The use of nested scopes is needed if within a particular task, there is a subtask that repeatedly allocates data with non-overlapping lifetimes. Nested scopes complicate reasoning about correctness - thus, when possible they are avoided. Dynamically enforced safety rules check that a memory scope with a longer lifetime does not hold a reference to an object allocated in a memory scope

with a shorter lifetime. This means that a scoped memory region may not hold a reference to an object allocated in an inner (more deeply nested) scope.

Programming with scoped memory, however, entails a loss of compositionality. This is because scoped memory adds an extra dimension to memory allocations – *where* an object has been allocated. This complicates reasoning about program correctness. For instance, components, when tested in one memory context, may work correctly, but may break when put in a particular scoped memory context. Consequently, code that has to run in scoped memory requires greater care to prevent errors than normal heap-allocated code. It is therefore also not unusual to find versions of a class specialized for use in scoped memory - and possibly specialized for a particular arrangement of scoped memory regions.

The RTSJ also introduces two new kinds of threads: real-time threads (`RealtimeThread`) that may access scoped memory regions; and no heap real-time threads (`NoHeapRealtimeThread`), which are extensions of real-time threads protected from garbage collection latencies. Much like Reflexes, the `NoHeapRealtimeThread` achieves this latency avoidance by executing in a memory region that is out of reach from the garbage collector, and from within here being restricted from referencing any heap-allocated objects. This restriction follows from the ability of the real-time threads to preempt the garbage collector, introducing a number of potential memory related problems, such as the risk of having dangling pointers, or observing objects in an inconsistent state as they are, for example, when being copied by a garbage collector.

Enforcing the restriction of not referencing any heap-allocated objects poses a fundamental problem with the `NoHeapRealtimeThread`. With `NoHeapRealtimeThread` the unit of restriction is the thread itself. Being a thread, the `NoHeapRealtimeThread` is allowed to do everything, and thus it is impossible to constrain it statically. Consequently, with the lack of any static ways of restricting the thread, a number of dynamic checks must be applied by the virtual machine to enforce this restriction. Specifically, the compiler inserts checks such that each read from or assignment to a field of reference type by the `NoHeapRealtimeThread` is guarded by read- and write-barriers, which are basically simple prologues of code executed to ensure safety before the actual read/assignment operation. Inevitably, executing these checks upon each field read/assignment introduce an execution overhead [PV06]. Aiming for sub-millisecond execution intervals, avoiding such a run-time overhead is crucial. In comparison, with Reflexes, the type restrictions are enforced statically, thereby avoiding the need to apply any run-time checks. This is made possible from the fact that with Reflexes the thread executing the task is not manipulable. Instead, the unit of restriction is the task itself rather than the thread, whereby it is easier to statically enforce restrictions. In fact, of the existing restricted programming models, only the RTSJ actually uses the thread as the unit of restriction.

Despite being shielded away from referencing heap-allocated objects, and executing a separate memory region, the `NoHeapRealtimeThread` is still subject to problem of lacking isolation between real-time and non-real-time parts of the system. Hence, a `NoHeapRealtimeThread` can block on acquiring a monitor held by an ordinary Java thread. For instance, this can happen if both the `NoHeapRealtimeThread` as well as the ordinary Java thread have synchronized access to some shared data allocated in the RTSJ-defined `ImmortalMemory` region, in which allocated objects can be referenced by both threads. To make matters worse, should the ordinary

Java thread happen to trigger a garbage collection, while the `NoHeapRealtimeThread` is being blocked, the `NoHeapRealtimeThread` may experience unbounded blocking.

2.4.2 Eventrons

Achieving sub-millisecond response time in Java has been the topic of numerous research papers, and several programming abstractions have been proposed with this objective in mind. Eventrons [SAB+06] is one such proposal for a new programming abstraction, which successfully manages to bring high responsiveness to real-time Java. Eventrons provide a simple programming model that is both efficient and provably safe. Contrary to the `NoHeapRealtimeThread`, Eventrons defines a *task* as the restricted unit, rather that the thread executing it.

One area that differentiates the various approaches relates to the trade-off between expressiveness and safety of memory operations. Static safety often comes at the expense of expressive power. The RTSJ defines an expressive API which supports many different real-time programming styles. However, with RTSJ any operation on a reference variable can result in an exception, and since no static checks are applied to prevent such errors, RTSJ relies on a number of expensive dynamic run-time checks. In contrast, both Reflexes and Eventrons provide stronger memory safety guarantees; once a program is successfully verified, no memory related errors will ever occur. Following from this, both Eventrons and Reflexes can be expected to outperform implementations of the RTSJ as they do not need run-time checks on reads/writes of references.

With the cost of increased predictability at the expense of expressive power, Eventrons are rather restrictive and introduce a number of significant restrictions to the Java language and the RTSJ APIs: they prevent allocation and mutation of any pointer structure. Reflexes fall in between; they are strictly more expressive than Eventrons but clearly less so than RTSJ. Specifically, Reflexes are executed in standard scoped memory enabling both allocation as well as pointer manipulation in so far that the latter complies with the standard scoped memory policies.

Much like Reflexes, Eventrons circumvent interference with the collector and allow user code to arbitrarily preempt the garbage collector. This is safe if: (1) the code does not allocate, (2) performs no blocking operations, and (3) that in the part of the heap accessed by the Eventrons code all reference fields are immutable and stay fixed through the execution. These safety properties are validated per Eventron using a powerful inter-procedural static analysis carried out precisely at application initialization-time, as it requires access to the in-memory data structures manipulated by each Eventron. Once validated, Eventrons do not require further run-time checks on memory access operations (as opposed to the `NoHeapRealtimeThread` in RTSJ).

In terms of development productivity, whereas correctness for Eventrons is only ascertained after deployment, the Reflex approach to static safety is arguably better as it relies on a small extension to the Java type system. Any safety problems found in the code are thus detected during development and reported to the programmer, rather than being deferred to run-time. Finally, Reflexes improve on Eventrons in term of startup times and JVM footprint as Eventrons must perform data flow analysis and compilation of the bytecode at startup. Oppositely, one

could reasonably argue that with Reflexes there is an issue of trust, since the Reflex code is only verified for type-safety statically; not at startup-time. In other words, any malicious changed made to Reflex code between compile time and run-time would not be detected nor prevented.

2.4.3 Exotasks

Exotasks [ABI+07] is another restricted programming model that extends Eventrons on a number of accounts. Most importantly, Exotasks attempt to tackle the interesting problem of how to ensure time-portability for real-time applications across platforms. Exotasks addresses this issue using a scheduling policy based on logical execution time (LET) inspired by Giotto [HHK01].

As its name implies, Exotasks are (like Eventrons and Reflexes) task-oriented. Where Eventrons are independent tasks only, Exotasks are organized in a graph connected by communication channels. Unlike Reflexes, which use a restricted *capsule* type to achieve zero copy communication-by-reference between tasks, Exotasks only allow for communication between tasks through scalar values or by deep copying objects passed over the channels between Exotasks. Whereas in Eventrons ordinary Java threads could communicate with Eventron tasks through scalar values, Exotasks completely prohibit this external communication. In contrast, with Reflexes external communication with ordinary Java threads is possible through scalars and transactional methods.

In terms of memory management, Exotasks, like Reflexes, are less restrictive than Eventrons in that they are permitted to perform allocation in their private memory regions during run-time. To manage memory allocations here, each private memory region of the Exotasks is maintained by a private garbage collector. The collector is scheduled to run in succession to each execution of the Exotask, assuming there is garbage to collect, thereby not causing any interference between the time-critical code and the garbage collector. In contrast, Reflexes do not have garbage collection of the private memory region. Instead, a separation of objects according to their lifetime is proposed, where an object then either lives for the lifetime of the Reflex instance itself, or for a periodic execution only.

2.4.4 The Ravenscar Profile for Ada

Restricted programming models are by no means exclusive to the Java language. The Ravenscar Profile [Bur99] is an example of a restricted programming model for Ada designed for safety-critical hard real-time systems. It has been standardized as part of Ada 2005 providing a subset of the Real-time Annex of the Ada language standard [ANS].

The main purpose of this language subset is to remove high overhead or complex features to improve memory and execution time efficiency, and to increase reliability and predictability, by removing non-deterministic and non-analyzable features. The programs following these restrictions follow a computational model that is deterministic and predictable, enabling static analysis in terms of functionality and timeliness before execution, e.g, schedulability analysis and calculations of bounds on memory-usage.

Applications consist of a number of tasks, communicating through protected objects. With applications declaring `Profile(ravenscar)` pragma, a set of restrictions are enforced by the compiler on the application code. These restrictions mostly relate to tasks, protected objects and delay statements.

The Ravenscar Profile mandates a scheduling model based on a priority preemptive scheduling, where each task is assigned a priority. This scheduling model is designed to minimize the upper bound on blocking time, to prevent deadlocks, and be verifiable that there is sufficient processing power available to ensure that all critical tasks meet their deadlines. Finally, the scheduling model assumes a single processor only [Rui05].

In terms of memory management, the Ravenscar Profile prohibits any dynamic allocation after system initialization time for the purpose of schedulability analysis. Consequently, the set of tasks in the application, and the *protected objects* with which the tasks communicate, must be known at system initialization time. Knowing these sets, the memory requirements of executing the task sets can be calculated and bounded. As mentioned earlier, Ada implementations rarely provide any garbage collection affecting predictability, a crucial property for safety-critical applications. Furthermore, the Ravenscar Profile prevents tasks from acquiring dynamic memory from the standard storage pool as this usage cannot be statically established [ANS]. Rather, aggregate data-structures must either be declared globally, such that the memory requirements can be calculated at initialization time, or the required storage must be allocated on the task stacks. Finally, tasks and protected objects are also prohibited from being deallocated.

Besides requiring fixed sets of tasks and protected objects, other restrictions are also applied to how tasks communicate. More specifically, a static synchronization and communication model is required, where tasks only communicate using protected objects, as opposed to using so-called *rendezvous* – the basic mechanism for synchronization and communication of Ada tasks. The protected objects have monitor-like behavior for ensuring sequential access to these objects. The Ravenscar Profile enforces that having entered such a monitor, the task is restricted from performing any operations that could cause the task to become internally suspended, which could cause other tasks to be blocked on the object. Furthermore, the Ravenscar Profile prevents task deadlocks on accessing such protected objects by requiring that protected objects are assigned a *ceiling priority* that is higher than the priorities of the tasks calling them.

Part II

Programming Model

3

The Reflex Programming Model

This chapter provides an overview of the Reflex programming model, introduces the basic concepts of the model, including the API, memory management and scheduling, and finishes off by demonstrating an example of a challenging real-time network Intrusion Detection System.

3.1 Overview

The *Reflex* programming model is a simple, statically type-safe programming model facilitating the construction of highly-responsive applications in the Java programming language.

Reflexes provide programming abstractions that makes it easy write and integrate simple periodic tasks or complex stream processors, both observing real-time timing constraints in the sub-millisecond range, into larger Java applications, and provide means for type-safe, obstruction-free interaction between the two. The Reflex programming model is non-intrusive in that it does not require changes to the standard Java syntax, nor does it require refactoring of existing code, permitting a high degree of reuse of standard libraries and legacy code.

Type-safety is achieved through a set of static safety checks inspired by an implicit ownership type system imposing a number of restrictions to the standard Java type system. The restrictions ensure isolation of time-critical computational activities – thus allowing multiple activities to run in parallel without blocking or experiencing data races – and provides a type-safe region-based memory model that permits these activities to compute even when the garbage collector is running, thereby circumventing garbage collection interference. These restrictions apply only to the time-critical activities, leaving the time-oblivious parts of the Java application unaffected, significantly increasing the ability to reuse and integrate legacy code.

The restrictions on the time-critical code are enforced statically at compile time through an extension of the standard Java compiler, mandating an extra pass over the successfully compiled Java source code. An interesting aspect of the restrictions is that although they put constraints on what the time-critical code can perform, they are not more restrictive than to allow time-critical code to be programmed using most of standard Java library classes. By enforcing the type restrictions statically rather than during run-time, the Reflex run-time engine can bypass the expensive run-time checks by the virtual machine, thus avoiding any negative impact on the performance of the real-time threads, a significant drawback of `NoHeapRealtimeThread` in the RTSJ.

Finally, Reflexes relies on a minimal extension set to the virtual machine, providing support for region-based memory, preemptible atomic regions [MBC+05], and priority preemptive scheduling.

3.2 Design Criteria

The inspiration to the Reflex programming model comes from the RTSJ. At first glance, one might wonder what added value that the Reflex programming model brings over and above RTSJ's `NoHeapRealtimeThread`, which, after all, is designed explicitly for real-time tasks with minimal latency requirements, and which is supported by all Java real-time virtual machines. However, as reported extensively in the literature, experiences implementing [BR01, CC03, PV03, ABC+06] and using [BCC+03, NB03, BN03, PFHV04, PV06] RTSJ have revealed a number of serious deficiencies. In particular, these deficiencies center around the scoped memory model of RTSJ, as described earlier.

With these deficiencies in mind, we have pursued a restricted programming model driven by the following design criteria:

3.2.1 Safety

A real-time program should never experience any memory related run-time errors. In contrast, with the RTSJ scoped memory model, and in particular in combination with the `NoHeapRealtimeThread`, any operation on a reference variable can result in a memory related run-time exception.

Inevitably, preventing such errors from occurring during run-time means that the programming model has to rely on some form of static checking to eliminate such potential risks before the program starts. For this to be feasible, the programming model must facilitate such checking, and this reveals one of the core problems of the `NoHeapRealtimeThread` construct of the RTSJ. In RTSJ the restricted entity is the `NoHeapRealtimeThread`. Being a thread (specifically, a subclass of `java.lang.Thread`), the `NoHeapRealtimeThread` can do anything, and thus restricting it statically is impossible.

In contrast, Reflexes as well as both Eventrons and Exotasks propose alternative entities of

restriction. For instance, the Reflex programming model proposes a `ReflexTask` construct as the restricted entity. Through applying restrictions on these entities, the Reflex programming model as well as both Eventrons and Exotasks provide stronger run-time guarantees than RTSJ; once a program has successfully been verified according to the safety checks, errors relating to referential integrity are guaranteed not to occur.

3.2.2 Expressiveness

Guaranteeing static safety in the programming model by restrictions comes at the expense of expressive power. Whereas the RTSJ has a rich API that supports many different styles of doing real-time programming, Eventrons is rather restrictive, for instance, prohibiting the program from doing any dynamic allocation and mutation of reference types. Reflexes and Exotasks fall in between; they are strictly more expressive than Eventrons, both permitting limited allocation and manipulation of reference types, but clearly less so than RTSJ.

The balance here is how to restrict the programming model just sufficiently, and in a general manner, in order to ensure the static safety guarantees without concurrently limiting the expressiveness to such a degree that programming with it becomes inconvenient, hindering adoption, or even rendered useless for certain types of real-time applications.

In the end, the true measure for the right level of expressiveness is a large base of applications successfully implemented with the use of the programming model in question. Unfortunately, there are only a handful of RTSJ programs available in open source form. Nevertheless, we have successfully implemented a variety of applications ranging from applications with a single, periodic real-time task, such as an audio-processing application converted from Eventrons [SAB+06], and a more realistic avionics collision detector application, to more complex real-time stream processors, such as a network intrusion detector system inspired by [SGVS99] and various digital signal processing applications, such as a filter bank calculation converted from [TKA02].

3.2.3 Simplicity

Correctness is often correlated with simplicity. In that respect both Eventrons, Exotasks and Reflexes provide programming models that are simpler and easier to use than RTSJ, and following from this, the correctness of the applications is easier to establish. For instance, an RTSJ program might run for hours before reaching a piece of code violating the memory integrity of the scoped memory model. Reproducing and debugging such violations is cumbersome and often non-trivial.

In contrast, the Reflex approach to static safety is arguably better as it relies on a small extension to the Java type system. With this approach, a Reflex program performing illegal memory operations will be caught early in the development phase, resulting only in compiler errors with direct references to the violating statements in the Java source code. Correctness for Eventrons is only ascertained after deployment, and while this is certainly better than with RTSJ, it may be harder for end-users to interpret error messages produced by a sophisticated program analysis.

3.2.4 Efficiency

Benefitting from the fact that the correctness of the real-time programs is established prior to actual run-time, Eventrons, Exotasks and Reflex programs are expected to outperform implementations of RTSJ as they do not need expensive run-time checks in the form of non-trivial read and write barriers around all reference operations to check for potential violations. Not having to apply these run-time checks, a source to a significant execution overhead [PV06] can be avoided.

Likewise, through a combination of static safety restrictions and support for obstruction-free interaction between ordinary Java threads and real-time tasks, Reflexes do not have to support priority inversion avoidance, and, finally, Reflexes have simpler memory region semantics than RTSJ. Compared to Eventrons and Exotasks, the Reflex programming model provides improvements in term of startup times and virtual machine footprint as the others must perform data flow analysis and inspection of the bytecode at startup.

3.3 Programming with Reflexes

A Reflex program consists of a *graph* containing a non-empty collection of nodes connected in some topology through a number of edges. The nodes in the graph represent Reflex *tasks* and the edges represent unidirectional *communication channels* connected between two distinct tasks. This choice relates directly to graph-based modeling systems, such as Simulink [Sim] and Ptolemy [Lee03], that are often used to design real-time control systems, or to stream-based programming languages like StreamIt [TKA02]. Fortunately, a single task is just a degenerate case of a graph, and so selecting a graph-based approach does not result in any fundamental loss. A Reflex graph and its tasks are constructed as a standard Java program following standard Java programming conventions, and can thus be constructed using standard Java development tools, such as IDEs and compilers.

A Reflex graph can run in pure isolation on the virtual machine or as part of a larger Java application. Being integrated with a Java application, ordinary Java threads can interact with the Reflex graph by invoking special methods on one or more Reflex tasks in the graph. Fig. 3.1 illustrates a Reflex graph consisting of three Reflex tasks, each in their own memory region, interconnected with unidirectional communications channels and its integration and interaction with a plain Java application, represented through an ordinary Java thread.

The Reflex task acts as the basic computational unit in the graph, consisting of user-defined persistent data structures, typed input and output channels, an (implicit) trigger on channel states, and user-specific logic implementing the functional behavior of the task – the activity. In order to ensure low latency, each task lives in a partition of the virtual machine's memory outside of the control of the garbage collector. Furthermore, the tasks in a graph are executed with a priority higher than ordinary Java threads. This allows the Reflex scheduler to safely preempt any ordinary Java thread, including the garbage collector. Hence, Reflex tasks can run without fear of being blocked by the garbage collector. This memory partitioning also serves

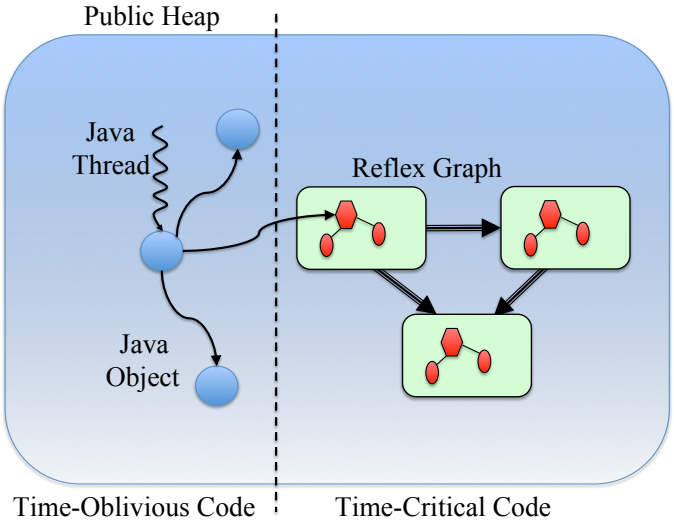


Figure 3.1: Illustration of a memory representation of a Java application consisting of a time-oblivious and a time-critical Reflex Graph. The Reflex Graph consists of three inter-connected Reflex tasks allocated in separate private memory areas, and its integration and interaction with normal, time-oblivious Java code.

to prevent synchronization hazards, such as a task blocking on a lock held by an ordinary Java thread, which in turn can by blocked by the garbage collector. Instead, transactional methods are used for non-blocking synchronization when Java threads need to communicate with tasks within the Reflex graph.

In terms of memory management, a Reflex graph is composed of three kinds of objects: *stable* objects, *transient* objects and *capsules*. Stable objects include the task instance itself and its internal state, their lifetime is equal to the lifetime of the task. Transient objects live as long as the execution of the task's activity. This split is motivated by real-time programming practices where a periodic task has persistent data and also manipulates temporary data structures that are reclaimed when it completes the current period's work. Finally, capsules are data objects used as messages between tasks, and are severely restricted in which data types they can carry. The capsule objects are managed by the Reflex run-time engine in a separate communication area. Specifying whether an object is stable, transient or capsule is done at the class level.

The Reflex run-time environment can execute multiple disjoint Reflex graphs concurrently. The run-time engine includes a scheduler that is responsible for releasing tasks. While a task can become schedulable any time new data appears on one of its associated input channels, the scheduler does not make guarantees of timeliness. It only ensures that each task will eventually be released. If the programmer requires timely execution of tasks, *clocks* must be used. When a task is connected to a clock, the scheduler arranges for the target task to be released according to the provided period. Hence, with this scheme a simple periodic activity would be modeled in a Reflex graph as a Reflex task connected to a clock. While multiple threads can drive the

execution of tasks in the graph, Reflexes ensures that the individual tasks are single-threaded.

3.3.1 Reflex Graph

A Reflex graph is constructed by extending the built-in abstract `ReflexGraph` class, and the programmer must implement one or more constructors. Fig. 3.2 shows an excerpt of the `ReflexGraph` class. The constructor is responsible of creating the tasks in the graph and connecting them according to the desired target topology. Once a graph is fully constructed, the constructor must invoke the `validate` method to check the graph. The validation, which is performed at run-time, only checks the well-formedness of the graph, and is not concerned with referential integrity of memory operations; those checks are performed statically during compile-time, and have at this point been established.

```
public abstract class ReflexGraph {

  public ReflexGraph() {...}
  public ReflexGraph(int priority) {...}
  public ReflexGraph(int priority, int commAreaSize) {...}

  public final void start() {...};
  public final void stop() {...}
  protected final void validate() throws ValidationException {...}

  protected final ReflexTask createTask(Class taskClass) {...}
  protected final ReflexTask createTask(Class taskClass, int stableSize, int transientSize) {...}
  protected final Clock createClock(int periodInMicrosecs) {...}
  protected final ReflexTask createRRSplitter(int count) {...}
  protected final ReflexTask createDupSplitter() {...}
  protected final ReflexTask createJoiner(int count) {...}

  protected final void connect(Clock source, ReflexTask target, String targetField) {...}
  protected final void connect(ReflexTask source, String sourceField,
                               ReflexTask target, String targetField, int size) {...}
  protected final void connect(ReflexTask source, String sourceField,
                               ReflexTask target, String targetField, int size, int rate) {...}
}
```

Figure 3.2: An excerpt of the abstract `ReflexGraph` class to be subclassed by the programmer in order to create and connect tasks in the graph according to user-specific requirements.

The graph validation involves (1) verifying that all channels are connected to fields in tasks expecting the same input/output types, (2) that there is sufficient space available within the private memory areas of the tasks and the communication area, if needed, and (3) that clocks are configured with periods supported by the underlying virtual machine.[1] Note, in this validation,

[1]Most operating systems can provide periods in the millisecond range. In our experiments, we use a release of Linux where the kernel has been patched to provide microsecond periods.

cyclic graphs do not pose a problem as they do not necessarily run indefinitely, e.g., if a task outputs no elements on its output channel(s). If during validation, any violations to the well-formedness of the graph are found, the `validate` method will throw a `ValidationException`.

From the point in time where the graph has passed the validation checks, the topology of the graph will be fixed throughout the lifetime of the graph. Hereafter, the `start` method can be invoked, causing the tasks in the graph to be scheduled. Any attempt to the invoke `start` method on a graph that either has not been validated at all, or did not pass the validation, will cause for an unchecked exception to be thrown.

The base `ReflexGraph` class provides methods for the reflective creation of tasks and channels, including clocks, as well as connecting the tasks. Reflection is needed because the creation of both channels and tasks must be allocated in the specific memory areas, it would be unsafe to allocate any of these objects in the heap as they would then be reachable to the garbage collector.

Finally, in order to enable communication between tasks in the graph, the `ReflexGraph` class provides a base constructor that causes for the creation of a communication area having the size of the provided constructor argument.

3.3.2 Reflex Task

The task is the computational unit in a Reflex graph, and is constructed by extending the built-in abstract `ReflexTask` class. The `ReflexTask` is executed by a real-time thread that is running at a higher priority than any other thread, including the garbage collector, in a priority preemptive manner. More specifically, the task is executed by a real-time thread having the priority provided to the `ReflexGraph`.

The programmer is responsible for providing an implementation of the abstract `execute` method, which defines the functional behavior of the task. Fig. 3.3 shows an excerpt of the `ReflexTask` class.

```
public abstract class ReflexTask implements Stable {

  public ReflexTask(int transientSize, int stableSize) {...}
  public abstract void execute();
  public void initialize() {}
  protected final Capsule makeCapsule(Class c) {...}
}
```

Figure 3.3: An excerpt of the abstract `ReflexTask` class to be subclassed by the programmer. The `ReflexTask` class is the computational unit in the Reflex graph, and its `execute` method must specify the user-specific functional behavior. The method `initialize` is declared with an empty body that can optionally be overridden.

The `execute` method of each task is invoked by the scheduler when the task is schedulable. By convention, the `execute` method is expected to eventually yield and give control back to the Reflex run-time environment – in most applications it would be a programming error for an activity to fail to terminate as this could block all tasks in the entire graph and cause serious deadline misses to occur.

A task is schedulable upon the arrival of data on one of its input channels according to the specified rate on the channels. More specifically, the rate specifies how much data the task needs on its individual input channels in order to execute. By default, for each channel this rate is set to one, but the programmer can optionally override the default input rate on the incoming channels, providing an alternative rate when a task is schedulable.

Finally, the `ReflexTask` class also declares a method `initialize`. The purpose of this method is to initialize the Reflex task, if necessary, before the tasks actually start. As such, the `initialize` method is invoked once during the lifetime of the task. The body of the `initialize` method is per default empty, but can optionally be overridden to provide task specific behavior. The method is invoked by the Reflex run-time engine on all tasks in the Reflex graph as the first thing when the `start` method on the `ReflexGraph` is invoked. Once the `initialize` methods have all been invoked in the graph, the clock task(s) in the Reflex graph can then be started.

3.3.3 Private Memory Region

Like other restricted programming models, Reflexes execute in complete isolation from the public heap garbage collector by letting the Reflex task run in its own heap-allocated private memory region, illustrated in Fig. 3.4. The `ReflexTask` instance itself is allocated within its private memory region to shield it away from the public heap garbage collector, while at the same time being reachable by ordinary Java threads, facilitating communication between the Reflex task and the plain Java application.

The memory region of a Reflex task is partitioned between a *stable heap* and a *transient area*. This separation is motivated by the requirement to keep latency at an absolute minimum in order to achieve the fastest response times possible. The sizes of both memory regions are chosen when instantiating the task, as reflected in Fig. 3.3. Since the size of the stable heap is fixed once the task is instantiated and the area is not garbage collected, allocations made in the stable heap must be managed carefully by the programmer to avoid `OutOfMemoryError`. The transient area is also fixed in size and serves as a per-execution allocation scratchpad. For the `execute` method, the default allocation context is always the transient area. Once the invocation of the `execute` method is complete, all allocations made in the transient area during its execution will be naively reclaimed in constant time, no expensive tracing is needed; any allocations made on the stable heap will remain. The strategy behind this design choice is that allocation of persistent state is the exception, and happens on the basis of computations and allocations made in the transient area.

Compared to this, in Exotasks [ABI+07] each task does not separate its private memory areas, but rather operates with stable heap, which is garbage collected either per schedule or on-

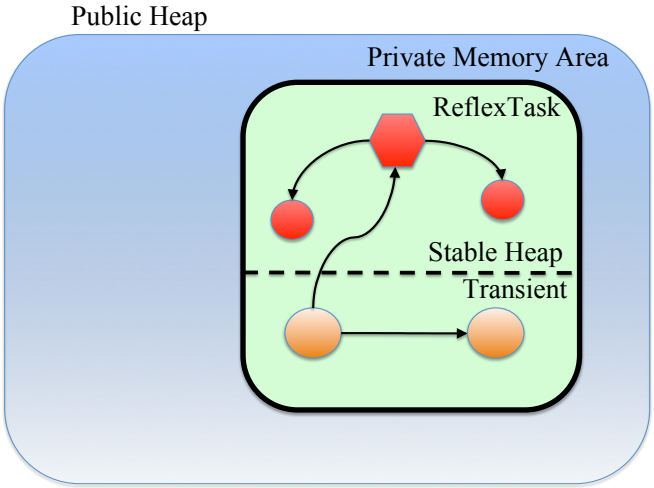


Figure 3.4: The memory model of a Reflex task enclosed in its own private memory region (green area) within the public heap. The figure illustrates a Reflex task (hexagon) in a private memory region with its object graphs of stable (red) and transient (orange) objects (circles).

demand. While this approach is attractive from a programming standpoint, as the programmer has to worry less about memory consumption, and efficient compared to a general garbage collection scheme, given the relatively small size of the stable heap and thus the root set to be collected, this approach might still be inappropriate for certain types of applications. For instance, for applications having minimal latency requirements, but which at the same time allocate a significant amount of data during each execution round to achieve a simple result set, this approach will push up the barrier on the fastest possible response times.

3.3.4 Object Lifetime Distinction

Following the separation of the private memory area, the Reflex programming model allows for a bimodal distribution of object lifetimes, and maintains two object graphs containing respectively stable and transient objects. Objects allocated in the stable heap live as long as the Reflex task itself, whereas objects allocated in the transient area only live for as long as the invocation of the task's `execute` method. This distinction in object lifetime fits well with the general execution pattern of periodic tasks, where data is allocated temporarily to generate the period's result that should survive several periods.

Specifying whether the allocation context for an object is the stable heap or the transient area of the Reflex task is done at the class level. By default, data allocated by a Reflex task is directed to the transient area. Only objects of classes implementing the `Stable` marker interface will be put on the stable heap and persist between invocations. Fig. 3.5 shows an example of a declaration of a stable class `Buffer`.

```
public class Buffer implements Stable {...}
```

Figure 3.5: An excerpt of the `Buffer` class showing its declaration as a `Stable` class, whose instance is to be allocated in the *stable* memory context of the task's own private memory area.

The `ReflexTask` base class is declared stable, as seen in Fig. 3.3, and will always be allocated in the stable heap of its own private memory area. Given the different lifetimes of the objects, for type safety reasons, stable objects are restricted to referencing only objects of stable type, whereas transient objects can reference both objects of transient and stable type. Referencing a transient object from a stable object could lead to a dangling pointer once the transient area has been reclaimed. Finally, allocations made by Reflex tasks are never directed to the public heap; either they occur in its transient area or in the stable heap.

By using class granularity for distinguishing between stable and transient objects, we relinquish the possibility of using the same class in both memory contexts. The alternative approach would be to introduce some per-object annotation, e.g., one could write code like `@stable HashMap h = @stable new HashMap()`. Unfortunately that is not sufficient. Specifically, the problem is that the code within the `HashMap` class may itself perform allocations, and those allocations would have to be destined in the exact same stable memory context to be consistent. Here, an approach treating the annotation as a type parameter, e.g. `new HashMap<@stable>()`, would help. However, this approach suffers a major drawback. While object granularity allows a greater degree of reuse, it is more heavyweight and requires retrofitting all library classes with generic parameters. The added effort and complexity does not seem warranted.

Another design choice is that the transient area is the default allocation context. Unlike for stable classes, transient classes have no restrictions on the types of their fields. This choice reflects the hypothesis that stable code is the smaller part of a Reflex and that it is less likely that we need to reuse legacy libraries in stable classes (part of the reason is that the allocation behavior of many library classes is not appropriate for an environment where objects are not reclaimed).

3.3.5 Stable Arrays

As mentioned, the default allocation context within a Reflex task is its transient area, except for classes explicitly marked stable. Following this design choice, primitive array objects allocated using statements such as `int[] ia = new int[10]` are thus always allocated in the transient area. As stable objects are restricted from referencing transient objects, it follows that stable objects cannot reference standard array objects.

To facilitate that stable classes can contain reference primitive array types, the Reflex API introduces a `StableArray` base class and provides a set of subclasses for each of the available primitive types. These `StableArray` classes encapsulate the different primitive arrays, and as their names imply, enable the allocation of these arrays in the stable heap. Fig. 3.6 shows

an excerpt of the abstract `StableArray` class. Besides its declaration as a stable class, its constructor is noteworthy, taking the primitive type and the size of the array type to be created and encapsulated.

```
public abstract class StableArray implements Stable {
  protected StableArray(Class type, int size) {...}
  protected final Object getArray() {...}
}
```

Figure 3.6: An excerpt of the `StableArray` base class for encapsulating arrays of primitive types. The constructor takes the primitive type and the size of the array to be created and encapsulated. The `getArray` method is used by the extending subclasses to access the encapsulated array.

Fig. 3.7 shows an excerpt of the `StableIntArray` class extending the `StableArray` class and facilitating the encapsulation of primitive integer arrays.

```
public class StableIntArray extends StableArray {
  public StableIntArray(int size) {
    super(int.class, size);
  }
  public void set(int index, int value) {
    ((int[]) getArray())[index] = value;
  }
  public int get(int index) {
    return ((int[]) getArray())[index];
  }
}
```

Figure 3.7: Example of a concrete extension of the `StableArray` base class for the encapsulation of primitive integer arrays.

Besides having the ability to create arrays of primitive types through the use of the `StableArray` classes, the Reflex programming model also enables the creation of arrays of stable reference type. In fact, if a class is declared stable then the array class derived from that class is simply considered stable too.

3.3.6 Task Exceptions

Given this object lifetime distinction, exception handling within a Reflex task requires special attention. When an exception is thrown within a Reflex task, the object is created with normal Java semantics. Given the default allocation context of the Reflex task, the exception object and its stack trace are created in the transient area within the Reflex task, and will thus be reclaimed like any other object following the completion of the invocation of the `execute` method. If the

exception propagates out of the `execute` method, the stack trace is printed and the thread is terminated.

3.3.7 Task Reclaiming

A Reflex task with its private memory region can be reclaimed by the public heap garbage collector once the entire graph to which it belongs is no longer active, and the task (and the graph it belongs to) is unreachable in the object graph of the Java application using it. Invoking the `stop` method on the `ReflexGraph` eventually causes the graph to become inactive.

3.4 Reflex Communication

Integrating a Reflex graph with real-time tasks into a time-oblivious Java application necessitates some form of communication means in order to share data. Likewise, the real-time tasks themselves might have a requirement to communicate internally in the graph. In time constrained systems using common practice synchronization operations to facilitate such communication is not appropriate and will cause for several problems.

The Reflex programming model enables type-safe, non-blocking communication as an alternative to using the common practice synchronization operations. The approach described facilitates two types of communication; (1) between the individual Reflex tasks in the graph – inter-task communication, and (2) between ordinary, time-oblivious Java threads and time-critical Reflex tasks – in both cases without causing the time-critical Reflex tasks to be blocked on a lock and miss deadlines.

3.4.1 Challenges Communicating between Tasks

The obvious challenge to be addressed to start with is how in the first place to share data between tasks that possibly can be executed by different real-time threads. Common programming practices enabling data to be shared safely use some sort of synchronization scheme based on the mutual exclusion. Java provides two different synchronization idioms, as illustrated in Fig. 3.8, based on either method synchronization or block synchronization within a method.

Looking apart from the general, obvious risk of experiencing deadlocks when using a lock-based scheme, for non-real-time applications such synchronization operations nevertheless make perfect sense as they have no explicit temporal requirements. In other words, the correctness criteria of such applications does not include the timing of the behavior. However, in real-time systems this mutually exclusion scheme can have some devastating side effects that could easily cause for temporal behavior of the tasks to be seriously compromised. Clearly, using such a lock-based scheme for sharing data with time-critical tasks is problematic.

Another challenge is what data types to allow to be communicated between tasks. One extreme would be to allow only primitive types, the approach adopted by the StreamIt project [TKA02].

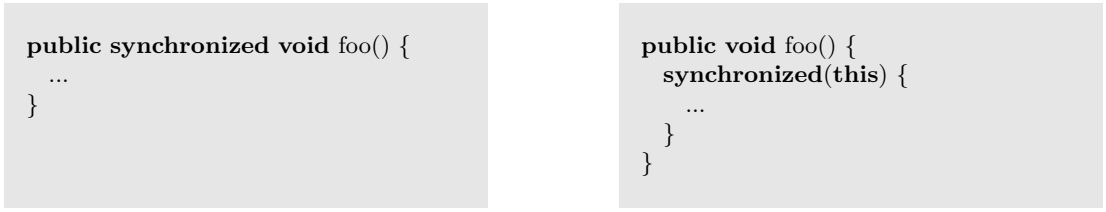


Figure 3.8: The synchronization idioms of the Java programming language (a) method synchronization, and (b) block synchronization.

The advantage of allowing primitive types only is that one does not need to worry about memory management or aliasing. On the other hand, if one wants to send complex sets of numbers, they would have to be sent individually in some specific order. While this may be acceptable in the case of simple data, encoding richer data structures is likely to be cumbersome for the programmer. However, there are good reasons for restricting the data types transferred between tasks. As soon as one adds objects to the computational model, it is necessary to provide support for their automatic memory management. The problem is compounded if garbage collection pauses are to be avoided. For instance, imagine a task retaining a reference to an object and sending a reference to the object to another task. When is it safe to reclaim that object? There is no obvious way, short of garbage collection, of ensuring that the virtual machine will not run out of memory. One possible strategy for dealing with object communication between tasks would be to simply deep copy the objects, the approach adopted by Exotasks. In other words, rather than passing a reference to the object to be communicated, a fresh copy is made of it and passed to the receiving task. While this is clearly safe and easy to manage in terms of memory, the deep copying process is expensive and might consume non-trivial time.

3.4.2 Non-Blocking Channels

The inter-task communication in Reflexes is designed with a key requirement in mind; enabling non-blocking, zero copy messaging between the tasks. A Reflex task communicates with other tasks through non-blocking channels. A channel is a fixed-sized, typed buffer connecting two Reflex tasks. Reflexes supports primitive type channels (all of Java's primitive types) and time channels holding periodic time stamps. Furthermore, Reflex channels can also carry objects, though the set of objects is restricted to those of `Capsule` types. Any Reflex task in a graph is required to have at least one input channel, except for the special `Clock` task.

Fig. 3.9 gives an overview of the `CapsuleChannel` class which is straightforward. The `TimeChannel` class, also seen in Fig. 3.9, is different in order to avoid storing, potentially large, numbers of clock ticks. Hence, it has two methods, one to put a current clock tick in microseconds on the channel, and one to return the latest unread clock tick.

The operations performed on a task's channels during a given release are atomic. Once the task

```
public class CapsuleChannel extends Channel {
  public int size() {...}
  public put(Capsule val) {...}
  public Capsule take() {...}
  public Capsule peek(int i) {...}
}

public class TimeChannel extends Channel {
  public void putTime() {...}
  public double getTime() {...}
}
```

Figure 3.9: Excerpts of the `CapsuleChannel` and `TimeChannel` classes for transferring respectively `Capsule` type data and time-stamps between tasks.

starts executing, its channels are logically 'frozen', no other task is allowed to modify them. All changes to channels are published when the task successfully completes.

Channels are created upon connecting two `ReflexTask` instances using the `connect` method on the `ReflexGraph` class, as seen in Fig. 3.2. The method will create the channel with its provided size, and connect it to the two tasks using the provided references to the source and destination `ReflexTask` instances. Specifically, the channel is connected reflectively to the fields in the two tasks having the names provided as arguments to the method. By default, a channel has a rate of one, but can optionally be overridden. The rate specifies how many elements the channel should contain for the target `ReflexTask` instance to become schedulable.

Channels are allocated in a memory area separate from any of the Reflex tasks using them – the communication area, as depicted in Fig. 3.10. In the event that the channels carry capsules, the capsules are also allocated in this memory area. The memory area is, like the task's private memory area, free of interference from the public heap garbage collector. The memory area is fixed-sized, and so the programmer has to carefully scale both the size of the memory area as well as the number and sizes of the channels that it holds. While this at first appears limiting, the actual number of capsule types used in an application as well as the instances created of each type, in our experience, are typically bounded, and thus adjusting the size of the communication area accordingly is feasible. The actual allocation of the memory area is performed by the Reflex run-time as part of the instantiation of the `ReflexGraph`, see the constructor in Fig. 3.2.

The current version of Reflexes does not support growable channels and, in case of overflow, silently drops packets. Other policies have been considered but have not been implemented. Variable sized channels, for example, can be added if users are willing to take the chance that resize operation does not cause for out of memory in the memory area holding the channels.

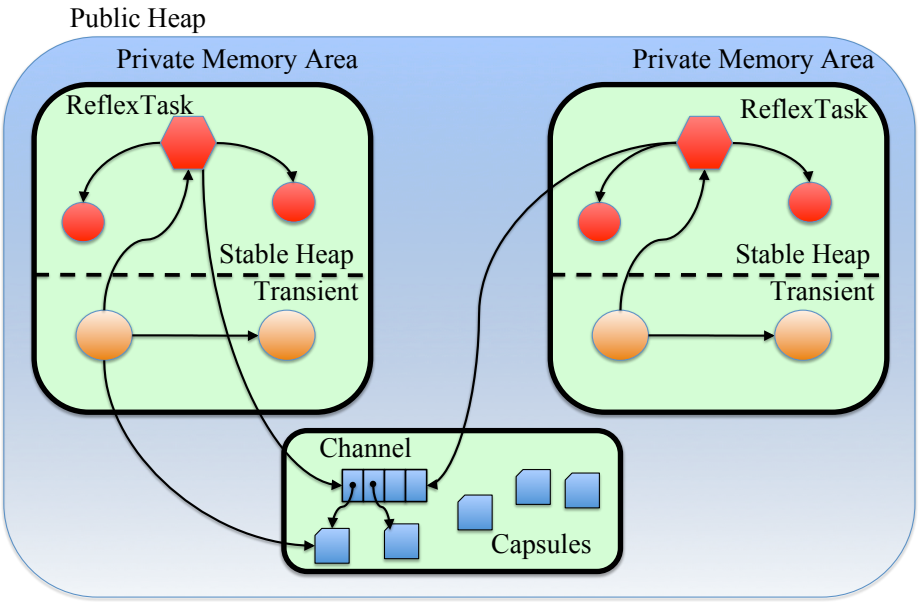


Figure 3.10: Reflex tasks communicate in zero copy style by passing on references to individual capsules. These references are pushed and popped from a channel shared by the two Reflexes. Both channels and capsules are allocated in a separate memory area managed by the Reflex run-time engine.

3.4.3 Capsules

Capsules, like channels, are designed with one key requirement: allow for type safe, zero copy communication between tasks in a Reflex graph. This seemingly simple requirement turns out to be challenging in a region-based system. Indeed, the question of where to allocate capsules, and when to deallocate them is a difficult one. They cannot be allocated in the transient memory of a task as they would be deallocated as soon as the task's `execute` method completes, leaving receiving tasks with a dangling pointer. Likewise, they should not be allocated in a task's stable memory as that area would quickly run out of space. Instead, as mentioned, the capsules are allocated from a pool managed by the Reflex run-time engine within the communication area. The invocation to the `makeCapsule` method on the `ReflexTask` class causes for a capsule of the provided type to be returned. If a free capsule of the given type is available in the pool it is returned. Otherwise, if possible, a capsule is instantiated in the pool by the Reflex run-time engine and returned. Reclamation of the capsule into the pool is automatic as soon as the capsule is not referenced by any task or channel.

There is, however, one case where copying capsules is unavoidable. If a task needs to put the same capsule on multiple output channels, zero copy semantics cannot be enforced. Rather, copies are created by the Reflex run-time engine when modifications to channels are published after the task's `execute` method returns.

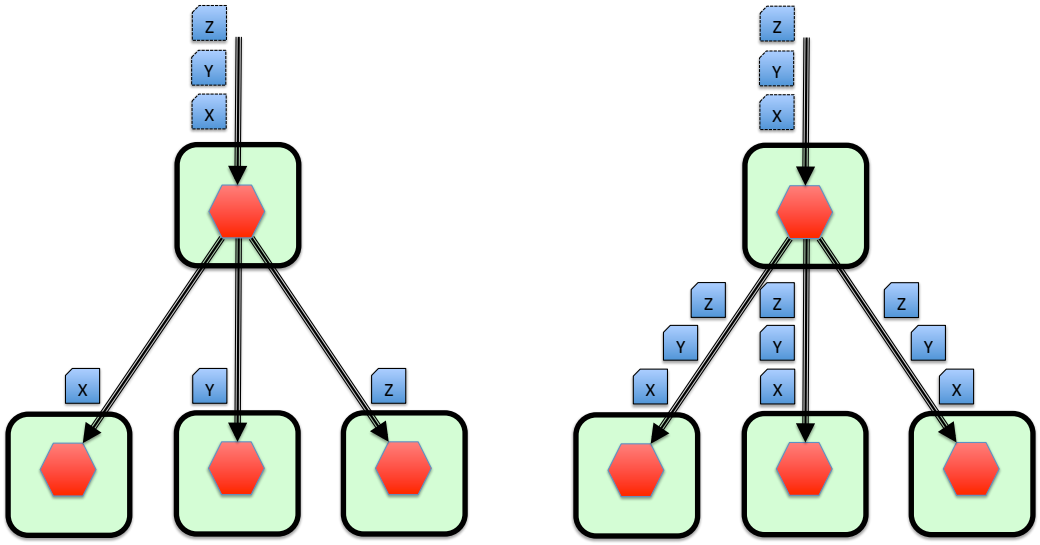
In order to guarantee memory safety, Reflexes impose severe restrictions on capsule classes. Specifically, to preserve isolation between tasks, a task must not retain a reference to a capsule that has been pushed to its output channel, and a capsule should not retain references to the task's stable data. In fact, the thing to notice is that a capsule should not be able to leak any references, i.e., it should be a reference-immutable construct. We address these requirements with a number of constraints that ensure that capsules can be used safely. For pragmatic reasons, however, we restrict the types carried by a capsule a bit further by limiting its field types to those that are statically determinable to be reference-immutable. Thus, we define a user-defined capsule to be a subclass of the `Capsule` class that can have fields which are restricted to either primitive types, or `final` unidimensional primitive array types. While these constraints have not proved too stifling so far, one could lift some of them if they prove to be too stringent, e.g., by allowing capsules to have `final` fields of any reference-immutable types. However, such easing of the constraints would come at the price of more complex set of static checks. Finally, capsules are maintained solely by the Reflex run-time engine, and the Reflex tasks are restricted from instantiating capsule instances. This way, we effectively prevent the capsule fields from being assigned reference type values stemming from inside a Reflex task.

3.4.4 Splitters and Joiners

Concerning communication between tasks, the Reflex API provides special convenience tasks for supporting data parallelization used in many stream applications. Inspired by StreamIt [TKA02], the Reflex programming model offers two built-in task types, a *splitter* and a *joiner*. The purpose of these special tasks is to enable implementations to multiplex/demultiplex data streams between Reflex tasks. More specifically, the splitter allows an input stream to be split into N output streams, and the joiner allows these streams to be merged again into a single output stream.

Both the splitter and joiner are created on the `ReflexGraph` class, as seen in Fig. 3.2. As illustrated by Fig. 3.11, the splitter comes with two different distribution policies; *Round-Robin*, and *Duplicate*. With the *Round-Robin* policy, the `SplitterTask` takes the number of elements provided in the `count` argument off its input channel and directs them to the next output channel, following a zero copy round-robin scheme. With the *Duplicate* policy a zero copy scheme is not possible as the same element has to go to all N output channels. Rather, the elements can be put on one output channel with zero copy semantics, but for the remaining *N-1* output channels copies of each element must be created and put on the channels. This copying is performed transparently by the Reflex run-time engine.

The `JoinerTask` only has round-robin semantics, merging N input streams into a single output stream by repeatedly taking the number of elements provided in the `count` argument off each input channel, one after the other, using zero copy semantics.



(a) Round-Robin (b) Duplicate

Figure 3.11: Examples of splitter tasks distributing messages on the input channel: (a) using a round-robin policy with `count` 1, and (b) using a duplicate policy.

3.4.5 Challenges Communicating with Ordinary Java Threads

Integrating time-critical tasks into a time-oblivious application raises some interesting challenges. Typically, in such an integrated system, threads are given priorities indicating their importance to the scheduler. Using a priority preemptive scheduler, the high priority, time-critical task will be able to preempt the ordinary, time-oblivious Java thread such that it can meet its temporal requirements. However, just as using lock-based synchronization to share data between time-critical tasks is problematic, as described earlier, the combination of thread priorities and lock-based synchronization schemes is equally problematic. Specifically, this combination might lead to situations of *priority inversion*, where the high-priority, time-critical task gets blocked and prevented from – in timely manner – accessing some shared data, to which a lower-priority thread holds the lock.

A number of well-known techniques exist to prevent priority inversion. For instance, in RTSJ every Java object has an internal lock that supports *priority inheritance*. With priority inheritance, a low-priority thread, T_L, holding a mutual exclusion lock, L_1, to some shared data will have its priority raised to that of a high-priority thread, T_H, if T_H gets blocked on L_1. However, assume now that T_L itself is blocked on another mutual exclusive lock, L_2. With priority inheritance, the thread holding the lock L_2 too must have its priority raised accordingly. In other words, to get out of a priority inversion situation, the raising of the priority to that of the high-priority thread, T_H, must be applied transitively to any lower-priority thread holding a lock waited for by T_H. Consequently, given the extensive use of locking in common Java programming practices, including the frequent usage hereof in the standard libraries, performing such an operation at run-time can easily lead to a non-negligible run-time overhead.

Besides the problem of how to synchronize access to shared data, communicating with reference types between Reflex tasks and ordinary Java threads also contradicts the isolation requirements of the Reflex memory model, and introduces issues that might compromise integrity of the memory region. As mentioned earlier, Reflex tasks avoid interference from the garbage collector by being isolated in separate private memory areas, and thereby not having any references to/from heap-allocated objects. However, prohibiting any reference types from passing the memory area boundaries would effectively disable any non-primitive type communication. Thus, a challenge is how to relax the isolation requirements to allow for certain references to pass the boundaries, while concurrently ensuring that no other reference could leak in or out of the Reflex task causing type safety problems.

3.4.6 Obstruction-free Communication with Transactional Methods

Besides the evident ability to communicate safely with ordinary Java threads through primitive types, Reflexes prevent synchronous operations by replacing lock-based synchronization with an obstruction-free communication scheme based on a simple form of transactional memory. More specifically, Reflexes propose a scheme based on methods that have transactional semantics. The principal behind transactional methods is to let an ordinary Java thread invoke certain methods on the time-critical task to which it holds a reference. Once inside the transactional method, the ordinary Java thread can access the data it shares with the Reflex task. Fig. 3.12 illustrates how an ordinary Java thread invokes a transactional method on a Reflex task passing along a reference to a heap-allocated object containing data to be put into the shared data structure.

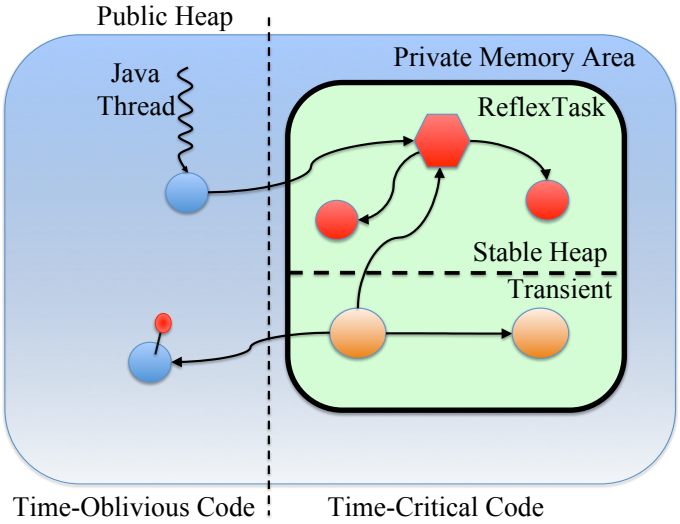


Figure 3.12: A time-oblivious, ordinary Java thread communicates with a Reflex task by invoking transactional methods directly on the `ReflexTask` instance. Transactional methods can pass in reference to heap-allocated objects (blue) that can be accessed from the default transient context in which the transactional method is executed.

In understanding the proposed transactional methods, it is useful to establish the semantics of transactional methods as follows. Any memory mutations made by the ordinary Java thread within the memory area of the Reflex task are only valid once the thread commits. If the ordinary Java thread runs until completion and commits, all its mutations are made durable in the private memory area of the Reflex task in which it invoked. More precisely, any transient object allocations made during the transactional method are reclaimed once exiting the method whereas mutations performed on stable objects are made durable. If, however, the Reflex task (or rather the real-time thread executing the task) is released by the scheduler and starts executing before the ordinary Java thread has committed, the ordinary Java thread is preempted, and any mutations made by the ordinary Java thread are rolled-back before the real-time thread will invoke the `execute` method of the `ReflexTask` instance. The ordinary Java thread will transparently abort its invocation and implicitly retry once the Reflex task has finished its execution. With this kind of transactional semantics, the time-critical Reflex task can run obstruction-free without missing deadlines, while the ordinary thread (hopefully) eventually will succeed with its invocation. Here, special care has to be taken by the programmer to avoid the theoretical possibility of the ordinary Java thread never completing its invocation, but rather retrying infinitely. Such a situation could happen if the time to complete the invocation of the transactional method is invariably longer than the period of a continuously executing Reflex task. Thus, the programmer must ensure that frequency of the time-critical task is such that, at some point, between two consecutive executions of a Reflex task there eventually would be enough idle time to allow for the ordinary Java thread to complete its transactional method invocation.

Although the use of transactional methods are obstruction-free in the sense that the time-critical task can respond immediately when it wants, and never will be blocked on an ordinary Java thread, they are not free of costs. In fact, the responsibility of performing the actual roll-back lies on the real-time thread, and thus adds a small run-time overhead with the complexity $\mathcal{O}(n)$, where n is the number of entries in the transaction log. As such, the programmer will have to take such a worst-case run-time overhead into account when choosing the frequency with which to run the Reflex task. Choosing a too short period could otherwise cause for deadline misses in the event that the Reflex task has to roll-back a transaction.

```
public class PacketReader extends ReflexTask {
  ...
  @atomic public void write(byte[] b) {...}
}
```

Figure 3.13: Example of declaration of method on `ReflexTask` class to be invoked with transactional semantics by ordinary Java threads.

Transactional methods to be invoked by ordinary Java threads are required to be declared on a subclass of the `ReflexTask` class in order to be reachable. To obtain transactional semantics on a method, the method signature must be annotated with `@atomic` as demonstrated with the `write` method in Fig. 3.13.

An interesting question at this point is, how can the time-critical task run free of garbage collection interference when the ordinary Java thread holds a direct reference to the `ReflexTask` instance and thus its object graph? In other words, since the object graph of the `ReflexTask` instance is reachable to the public heap garbage collector, is it not subject to interference? The answer to this question is really dependent on the virtual machine implementation. However, garbage collectors of RTSJ-compliant virtual machines typically perform range checks of pointers and will scope the scanning once a pointer points to something that is not on the public heap. We assume and rely on this property from the garbage collector. A garbage collector that would behave differently would invalidate our approach to garbage collector interference avoidance.

3.4.7 Method Argument Restrictions

For reasons of type-safety, Reflexes restrict the possible types of the parameters in the signature of the transactional methods to include only primitives and primitive array types. While this might seem rather restrictive, most examples of Reflex applications that we have been able to identify only deal with primitive values and arrays, thus we believe this is a sensible choice.

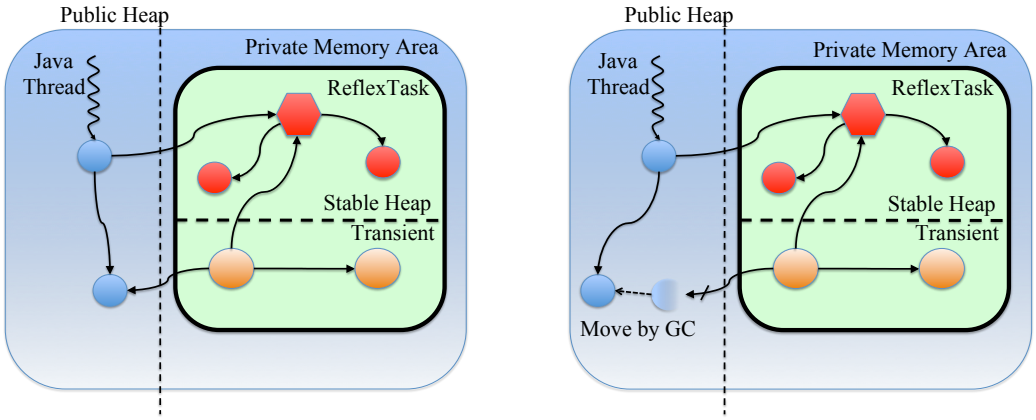
Special care has to be taken concerning the allowed heap-allocated arguments of primitive array types to be passed to the transactional methods, since in Java these are considered normal reference types. Though no references can leak when using a primitive type array, there could be a potential type-safety issue if a reference to the primitive array type were to be stored in a field in the Reflex task as such a reference would be unreachable to the public heap garbage collector. Consequently, if the garbage collector subsequently were to move the primitive array object on the public heap, e.g., for heap compaction purposes, the Reflex task would observe a dangling pointer since its reference would not be adjusted.

It turns out, however, that the stable/transient distinction of Reflexes prevents this from happening in the first place. Recall, all primitive array types are always considered of transient type, and thus can never be assigned to a field in the Reflex task, as Reflex tasks (as stable types) are restricted from having fields of anything but stable types. Nevertheless, a related problem remains concerning the type-safety of using primitive arrays in transactional methods.

Transactional methods execute in the transient area of the Reflex task, and as a consequence any references from here are out of reach from the public heap garbage collector. This poses a problem if an ordinary Java thread invokes a transactional method, bringing in as argument a reference to a heap-allocated primitive array object. Specifically, if during the invocation, the garbage collector preempts the invoking ordinary Java thread and moves the heap-allocated primitive array object, the ordinary thread's reference to that object will not be adjusted accordingly as it too is not reachable by the garbage collector. This problem is illustrated in Fig. 3.14.

To encounter this, we rely on assistance from the virtual machine by requiring that such method arguments are *pinned* to their location on the heap upon entering the transient area, as indicated by the *pinned* heap-allocated object in Fig. 3.12. Once the ordinary Java thread returns from the invocation of the transactional method, the arguments can safely be unpinned again.

The return type of a transactional method is even more restricted than its method arguments,



(a) (b)

Figure 3.14: Illustrating the problem of having the garbage collector moving the heap-allocated object used as argument for the transactional method. Transactional methods are executed in the transient area of the Reflex task, and any references from here are unreachable to the public heap garbage collector. Thus, if the garbage collector preempts an ordinary Java thread while invoking a transactional method, and moves a heap-allocated object, it cannot adjust any references to this object from within the transient area. (a) illustrates the references to the heap-allocated object from a transient one before the ordinary Java thread is preempted by the garbage collector. (b) illustrates how the garbage collector has moved the heap-allocated object referenced from within the transactional method, but not adjusted the reference to the object from within the transient area, leading to a dangling pointer.

allowing only primitive types. This further restriction is necessary first to prevent a heap-allocated object from observing a dangling pointer, if the returned reference would originate from a transient object that would be purged following the ordinary Java thread's exit from the transactional method, and second to prevent leaking of references to stable objects from within the Reflex task.

3.4.8 Communicating through Static Variables

In addition to transactional methods, Reflex tasks can communicate with ordinary Java threads (and for that matter among the tasks themselves) using static variables. In enforcing isolation of a Reflex task, static variables pose a particular type-safety problem as references to objects allocated in different Reflex tasks or on the heap, could easily pass the isolation boundaries through these. To circumvent this leaking, the Reflex programming model restricts the use of static variables to primitive and *reference-immutable* types. Informally, an object of reference-immutable type provides access to a graph of objects connected by references that recursively cannot change but containing objects whose elements can change, e.g., fields of primitive type [SAB+06].

Like for heap-allocated arguments passed to the transactional method, accessing a static heap-allocated variable from within the memory context of a Reflex task faces similar problems to that

illustrated in Fig. 3.14. If that public heap garbage collector was in the middle of moving the static variable when it got preempted by the Reflex task, the task could find itself with a dangling pointer, or observe the variable in an inconsistent state. Also here, we avoid experiencing such situations by requiring that heap-allocated static variables are pinned on the heap. However, unlike the arguments for the transactional methods, the static variables accessed by the tasks in the Reflex graph must be pinned for the entire lifetime of the graph. Fig. 3.15 illustrates how the Reflex task can access a pinned static variable located on the heap.

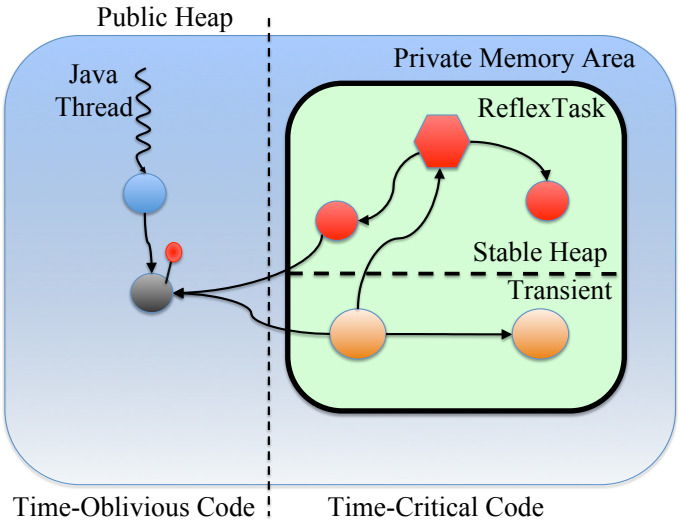


Figure 3.15: Communicating between ordinary Java threads and Reflex tasks through heap-allocated static variables (black) is permitted but restricted to primitive and reference-immutable types. Static variables of reference-immutable types are in addition required to be pinned to their heap location throughout the lifetime of the Reflex graph.

3.4.9 Choosing and Combining Communication Mechanisms

The Reflex programming model facilitates two different means of communication between ordinary Java threads and time-critical Reflex tasks, and one might rightly ask when using one type of communication is more appropriate than the other.

For the most part, the transactional methods option stresses precise knowledge of what was and was not communicated, and hence is particularly useful when there needs to be coordination (instead of traditional Java level synchronization) between tasks and ordinary Java threads. This is appropriate when the amount of data transferred into the restricted portion of the program is small.

On the other hand, a Reflex task's transactional methods can delay the restricted thread during a roll-back operation, and so this option is not a very good choice for bulk data transfer from normal to restricted threads. Combining the two options in the same program is thus a very

powerful feature of Reflexes; it is usually possible to partition the communication capabilities so that bulk data updates are done in a non-transactional way but notification and coordination of state changes are done transactionally.

3.4.10 Synchronization Operations

Though they might have a significant impact on thread scheduling, the Reflex programming model does not prevent the programmer from using synchronization operations nor the `wait/-notify` protocols within a Reflex task. However, Reflexes do not mandate priority inheritance semantics for locks, and thus it follows from here that avoiding priority inversion cannot be guaranteed. Hence, the use of synchronization operations within a Reflex are generally discouraged.

Nevertheless, the reason for not prohibiting them is that there might be situations where backward compatibility with library classes makes such usage unavoidable. Thus, while their usage is permitted, their consequences on scheduling remains non-deterministic. In contrast, the proposed approach to sharing data through transactional methods is type-safe and avoids problems of priority inversion.

3.5 Scheduling

When the unit of restriction is a graph of tasks rather than a thread, threads must be managed implicitly, and the executions of the individual tasks within the graph must be *scheduled*.

The Reflex programming model specifies a *time triggered* scheduling policy. This time triggered scheduling is embodied in the `Clock` task, which supports periodic execution of the Reflex graph. A Reflex graph is required to have one `Clock` task in order to execute. The `Clock` task is connected to a `TimeChannel` that again is connected to a `ReflexTask`, as depicted in Fig. 3.16. Upon triggering, the `Clock` task publishes a time stamp on its output time channel, causing for the `ReflexTask` reading off this channel to become schedulable.

Multiple graphs running on the same virtual machine are executed by multiple threads, each graph has a priority (where some graphs might have the same) that is set upon graph instantiation. Tasks in a graph are executed with a thread having the priority inherited by the graph. These threads are mapped directly to operating system threads.

Threads are not required to be mapped to tasks following a one-to-one scheme. However, as a minimum of threads are assigned to `Clock` tasks that then, within the period, simply traverse as far down in the graph possible and execute all schedulable tasks, a simple scheme that makes sense on a uni-processor machine. On a uni-processor platform, executing the Reflex graph using multiple threads would not contribute to any true parallelism, but rather extend the total execution time of the graph by introducing an execution overhead of context switching between the threads. Contrary, on a multi-processor machine applying multiple threads would be beneficial for purposes of parallelism as different threads are run by multiple processors.

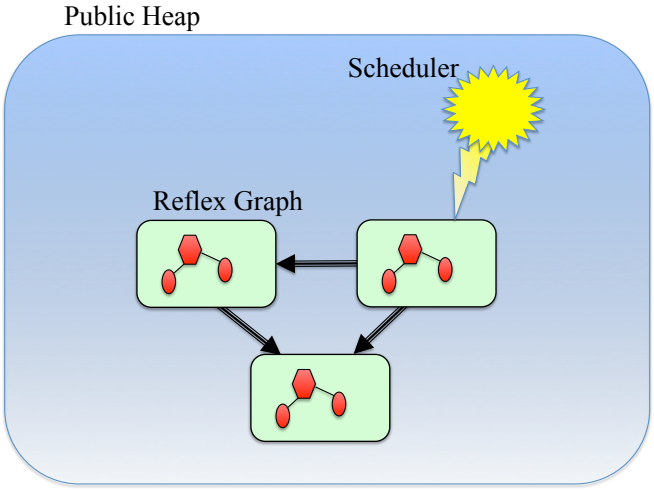


Figure 3.16: Each Reflex graph is triggered by a time triggered scheduler using the `Clock` task. Threads are not required to be assigned to task following a one-to-one scheme. As a minimum, a single thread is assigned to the `Clock` task that then traverses the Reflex graph and executes all schedulable tasks.

3.6 Example: Intrusion Detector System

To demonstrate the power and applicability of Reflexes on real-world applications, we have implemented a challenging real-time stream processing application in the form of an Intrusion Detection System (IDS), inspired by [SGVS99], which analyzes a stream of raw network packets and detects intrusions by pattern matching.

Stream processing is a programming paradigm suited for a class of *data driven* applications. In a stream processing language, a program is an acyclic graph composed of *tasks* connected by *data channels*. Each task in the graph is a functional unit that consumes data from its input channel and pushes results on its output channels. More specifically, the behavior of each task is entirely determined by the data on the channel and the task's internal state. This last property allows for parallel implementations, since task communication is restricted to channels, they can be scheduled in parallel.

The stream processing paradigm is a challenging application area given some of its key requirements [ScZ05]; (1) keep the data *moving* meaning that data should be processed with as little buffering as possible, and (2) the system should respond *instantaneously*, i.e., in a timely and responsive fashion, such that execution pauses that might lead to dropped data can be avoided.

Fig. 3.18 shows the declaration of the Reflex graph class `IDSGraph`, which instantiates and connects the tasks that combined implement the intrusion detection system. The argument to the `IDSGraph` constructor is the period in microseconds provided to the clock task in the graph. Fig. 3.17 provides a graphical illustration of the same Reflex graph, its tasks, and how the tasks

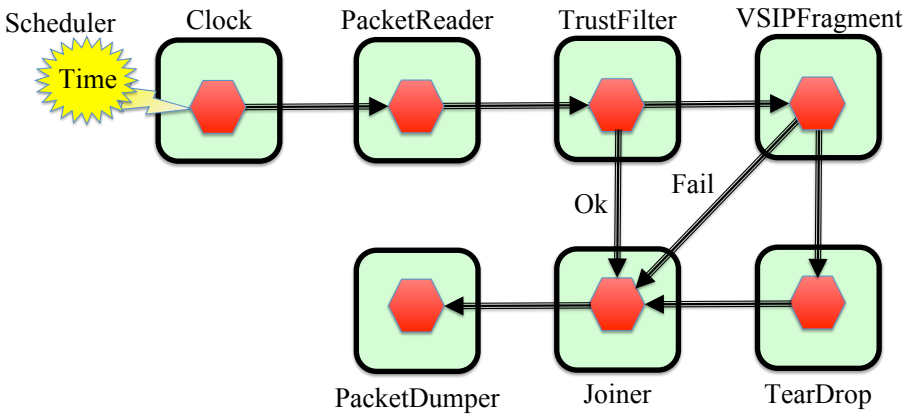


Figure 3.17: Graphical representation of the Reflex graph of an Intrusion Detection System consisting of six tasks and a clock task triggered periodically by a time triggered scheduler.

are connected.

The capsules being passed around the system represent different network packets: Ethernet, IP, TCP and UDP. Object-oriented techniques are useful in the implementation as nested structure of protocol headers are modeled by inheritance. For instance, the IP capsule class (`IP_Hdr`) is a subclass of the Ethernet capsule class (`Ether_Hdr` seen in Fig. 3.19) with extra fields to store IP protocol information.

Fig. 3.20 shows `PacketReader` class that creates capsules representing network packets from a raw stream of bytes. For our experiments, we simulate the network with the `Synthesizer` class. Fig. 3.21 illustrates the `main` method of the Intrusion Detector System application, showing the creation of the Reflex graph and the synthesizer. The synthesizer runs as an ordinary Java thread, and feeds the `PacketReader` task instance with a raw stream of bytes to be analyzed. Communication between the synthesizer and the `PacketReader` is done by invoking the `write` method on the `PacketReader`. This method takes a reference to a buffer of data (primitive byte array) allocated on the heap and parses it to create packets. The `write` method is annotated `@atomic` to give it transactional semantics, thereby ensuring that the task can safely preempt the synthesizer thread at any time.

The `PacketReader` buffers data in its stable memory with the `Buffer` class, shown in Fig. 3.22. Being referred from an instance field of the `PacketReader` task, the `Buffer` class itself is declared stable (by implementing the `Stable` interface), and in addition contains a primitive array of bytes. To satisfy the static safety constraints, we use the `StableByteArray` class to represent the primitive array within the stable class.

The reader uses the `readPacket` method to initialize capsules from the data stored in the buffer. The capsule instance itself in which to read the data is retrieved from the capsule pool through the `makeCapsule` call. The methods `startRead`, `commitRead`, and `abortRead` are used to ensure that only whole packets are read from the buffer. They do not need synchronization

```
public class IDSGraph extends ReflexGraph {
  private Clock clock;
  private PacketReader packetReader;
  private ReflexTask trustFilter, vsipFragment, tearDrop, joiner, packetDumper;

  public IDSGraph(int periodInMicrosecs)
    throws ValidationException {
    super(ReflexGraph.DEFAULT_PRIORITY, ReflexGraph.DEFAULT_COMMAREASIZE);
    clock = createClock(periodInMicrosecs);

    packetReader = (PacketReader) createTask(PacketReader.class);
    trustFilter = createTask(TrustFilter.class);
    vsipFragment = createTask(VSIPFragments.class);
    tearDrop = createTask(TearDrop.class);
    joiner = createTask(Joiner.class);
    packetDumper = createTask(PacketDumper.class);

    connect(clock, packetReader);
    connect(packetReader, trustFilter, 10);
    connect(trustFilter, vsipFragment, 10);
    connect(trustFilter, "ok", joiner, 10);
    connect(vsipFragment, "fail", joiner, 10);
    connect(vsipFragment, tearDrop, 10);
    connect(tearDrop, joiner, 10);
    connect(joiner, packetDumper, 10);

    validate();
  }
}
```

Figure 3.18: Implementing a `ReflexGraph` subclass. The `IDSGraph` class extends the abstract `ReflexGraph` class, declares a constructor for setting up the graph with default priority and communication area. Note, how at the end of the constructor the `validate` method is invoked, causing the graph to be validated.

since (1) potential higher priority tasks have no way to access the buffer (thanks to the isolation), and (2) ordinary Java threads, that can access the buffer through the `write` method, cannot preempt the Reflex task execution, assuming a priority-preemptive scheduling policy where the task runs at higher priorities than ordinary Java threads.

The packets first go to the `TrustFilter`, which looks for packets that match a trusted pattern; these packets will not require further analysis. Other packets are forwarded to the `VSIPFragment` task. This task detects IP fragments that are smaller than TCP headers. These are dangerous as they can be used to bypass packet-filtering firewalls. The `TearDrop` task recognizes attacks that involves IP packets that overlap.

The three tasks, `TrustFilter`, `VSIPFragment`, and `TearDrop` have a similar structure: an input channel (`in`) for incoming packets to analyze and two output channels, one for packets caught by

```
public class Ether_Hdr extends Capsule {
  ...
  public final int ETH_LEN = 6;
  final byte[] e_dst;
  final byte[] e_src;

  private Ether_Hdr() {
    e_dst = new byte[ETH_LEN];
    e_src = new byte[ETH_LEN];
  }
}
```

Figure 3.19: An excerpt of the `Ether_Hdr` capsule containing primitive byte arrays.

```
public class PacketReader extends ReflexTask {
  private Channel out;
  private Buffer buffer = new Buffer(16384);
  private int underruns;

  public void execute() {
    TCP_Hdr p = (TCP_Hdr) makeCapsule(TCP_Hdr.class);
    if (readPacket(p) < 0) underruns++;
    else out.put(p);
  }

  @atomic public void write(byte[] b) {
    buffer.write(b);
  }

  private int readPacket(TCP_Hdr p) {
    try {
      buffer.startRead();
      for (int i=0; i<Ether_Hdr.ETH_LEN; i++) p.e_dst[i] = buffer.read_8();
      ...
      return buffer.commitRead();
    }
    catch (UnderrunEx e) { buffer.abortRead(); ... }
  }
}
```

Figure 3.20: An excerpt of the `PacketReader` task that reads packets received from the ordinary Java thread and pushes them down in the graph. The `write` method, invoked by the ordinary Java thread, is declared to have transactional semantics. The ordinary Java thread and the `PacketReader` share a bounded buffer from which they respectively write and read.

the tasks (`ok` or `fail`), the other one for uncaught packets (`out`). These tasks also mark caught packets with meta-data that can be used in further treatment, logging or statistics. The task

```
public class IDSApplication {
  public static void main(String argv[]) {
    try {
      IDSGraph graph = new IDSGraph(PERIOD_MICROS);
      new Synthetizer(graph.getPacketReader()).start();
      graph.start();
    }
    catch (ValidationException e) {
      System.err.println("Graph failed validation: " + e.getMessage());
      System.exit(−1);
    }
  }
}
```

Figure 3.21: The main Java application instantiating the `IDSGraph` with some periodicity and creating the `Synthesizer` generating the packets, and interacting with the Reflex graph by invoking a transactional method on the `PacketReader` task.

```
public class Buffer implements Stable {
  private final StableByteArray data;
  private int pos, lastpos;

  public Buffer(int cap) {
    data = new StableByteArray(cap);
  }

  public int write(byte[] b) { ... }

  public void startRead() { lastpos = pos; }
  public int commitRead() { return pos−lastpos; }
  public void abortRead() { pos = lastpos; }

  public int read_32 throws UnderrunException { ... }
  public short read_16 throws UnderrunException { ... }
  public byte read_8 throws UnderrunException { ... }
}
```

Figure 3.22: An excerpt of the `Buffer` class shared by the ordinary Java thread and the `PacketReader` to exchange data. Note, that the class is declared stable as it is used as an instance field on the `PacketReader` task (which inherently is stable), and that as a stable class it uses the `StableByteArray` type to represent a primitive byte array.

implementations rely on an automaton stored in stable space to recognize patterns on packet sequences that correspond to attacks. For instance, the `VSIPFragment` (see Fig. 3.23) uses the pattern matcher shown in Fig. 3.24. The different pattern matchers used in this example all subclass the `PatternMatcher` class seen in Fig. 3.25.

```
public class VSIPFragment extends ReflexTask {
  private InCapsulePort in;
  private OutCapsulePort out;
  private OutCapsulePort fail;
  private VSIPFragmentMatcher pm = new VSIPFragmentMatcher();
  private TCPFragTable tcpFrag = new TCPFragTable();

  public void execute() {
    Ether_Hdr p = (Ether_Hdr) in.take();
    if (p instanceof TCP_Hdr) {
      TCP_Hdr t = (TCP_Hdr) p;
      if (pm.step(t)) {
        tcpFrag.inc(t.saddr);
        p.filtered = true;
        fail.put(p);
        return;
      }
    }
    out.put(p);
  }
}
```

Figure 3.23: An excerpt of the `VSIPFragment` class responsible for detecting IP fragments that are smaller than TCP headers. For this detection, it relies on a pattern matcher. Note, how the task in its `execute` method reads a `Ether_Hdr` packet from its input channel, and, depending on the result of the pattern matcher, puts the packet on different output channels for further processing, as also illustrated in Fig. 3.17.

The convenience task, `Joiner`, is used to transform a stream of data from multiple input tasks to a single stream of data. The last Reflex task, `PacketDumper`, gathers statistics of the whole intrusion detection process thanks to the meta-data written on packed by the previous tasks.

```
public class VSIPFragmentMatcher extends PatternMatcher {
  final int MY_NET = ...
  final int MY_NET_MASK = ...

  private boolean myNetAddr(int addr) {
    return (addr & MY_NET_MASK) == MY_NET;
  }

  private boolean isFrag(TCP_Hdr p) {
    return (p.more_frags) || (p.frag_offset != 0);
  }

  public VSIPFragmentMatcher() {
    final State rx = new State(this, true);
    initState.bind(rx, new Condition() {
      public boolean evaluate(Ether_Hdr e) {
        if (!(e instanceof TCP_Hdr)) return false;
        TCP_Hdr t = (TCP_Hdr) e;
        if (myNetAddr(t.daddr) && isFrag(t) && t.tot_len < 48)
          return true;
        return false;
      }
    });
  }
}
```

Figure 3.24: An excerpt of the `VSIPFragmentMatcher` class responsible for detecting small IP packets. Note how the `VSIPFragmentMatcher` is not itself declared stable; rather it inherits its stable property from the extension of the stable `PatternMatcher` class.

```
public abstract class PatternMatcher implements Stable {
  protected final State initState = new State(this, State.S_INIT, false);
  private State currentState = initState;

  static interface Condition extends Stable {
    public boolean evaluate(Ether_Hdr e);
  }

  static class Pair implements Stable {
    ...
  }

  State reset(Ether_Hdr e) {
    for (Iterator i = initState.transitions.iterator(); i.hasNext(); ) {
      Pair p = (Pair) i.next();
      if (p.c.evaluate(e)) return p.to;
    }
    return initState;
  }

  static class State implements Stable {
    final static int ST_INIT = ...
    final static int ST_NONAME = ...
    private int name;
    private boolean terminator;
    private StableList transitions = new StableList();
    private PatternMatcher pm;

    public State(PatternMatcher pm, boolean terminator) {
      this(pm, ST_NONAME, terminator);
    }

    public State(PatternMatcher pm, int name, boolean terminator) {
      ...
    }

    public State evaluate(Ether_Hdr e) {
      for (Iterator i = transitions.iterator(); i.hasNext(); ) {
        Pair p = (Pair) i.next();
        boolean res = p.c.evaluate(e);
        if (res) return p.to;
      }
      if (name == ST_INIT) return pm.getInitState();
      return pm.reset(e);
    }
    ...
  }

  public boolean step(Ether_Hdr e) {
    currentState = currentState.evaluate(e);
    return currentState.terminator;
  }
}
```

Figure 3.25: An excerpt of the general purpose `PatternMatcher` class used by several of the Reflex tasks in the IDS graph for pattern matching. Note, how it declares several stable types, which are used as field types in some of its instance fields.

4 Static Safety Checking

To avoid interference from the public heap garbage collector, Reflexes rely on strict isolation between Reflex tasks themselves, and between Reflex tasks and time-oblivious Java code. The goal of the safety checks is to statically guarantee this isolation by restricting unsafe code that would violate the memory integrity and allow access to heap-allocated objects in inconsistent states, and dangling pointers to be observed.

This chapter provides an informal specification of the set of static safety checks enforcing these isolation requirements, and discusses some of the design choices. Furthermore, we report on an experiment showing that the static checks are not more restrictive than allowing most of the Java collection framework to be used unmodified within a Reflex task.

4.1 Checking Principles

Reflexes use an approach inspired by previous work on ownership type systems [ZBH+08] to statically enforce the restrictions ensuring isolation and thus preventing dangling pointers and observing inconsistent heap-allocated objects. Ownership types were first proposed by Noble, Potter and Vitek in [NVP98] as a way to control aliasing in object-oriented systems. As with other ownership type systems [CPN98, BLR02, BSBR03], there is a notion of a explicit dominator declared on the class that encapsulates access to a subgraph of objects.

In contrast, the static safety checks used for Reflexes and presented here operate with *implicit* ownership in that no ownership parameters are to be specified on the class declarations. Instead, ownership is defaulted using straight-forward rules; every Reflex task encapsulates and owns all objects allocated within its private memory region. Given this ownership, the static safety checks

ensure that references to objects owned by a Reflex task are never accessed from outside, that the Reflex tasks cannot reference heap-allocated objects (except a few exceptions), and that stable objects cannot reference transient ones. Fig. 4.1 illustrates the legal and illegal object references between the time-critical and time-oblivious code to be respectively permitted and prohibited by a type checker enforcing the static safety checks.

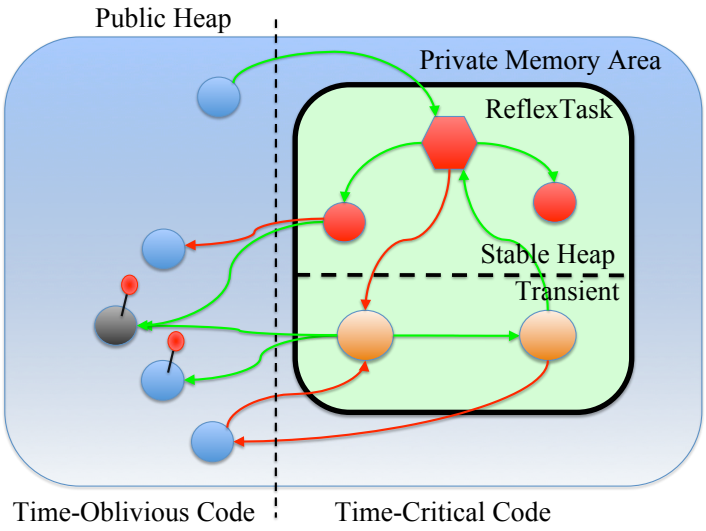


Figure 4.1: The legal and illegal object references in and out of a Reflex task that the static safety checks must ensure are respectively allowed and caught. The figure illustrates a `ReflexTask` in a private memory area with its object graphs of stable and transient objects as well as a number of heap-allocated objects and static variables, of which some are pinned. Object references are illustrated with green and red arrows, representing legal and illegal references respectively.

Most ownership type systems require fairly extensive changes to the application code, by annotating individual classes with one or more owner parameters. Much like Java generics, these parameters are expected to be erased at compile time. This approach, however, has an important drawback; it prevents reuse of legacy code, by requiring a complete refactoring of all library classes in order to annotate the owner, and does not interact well with raw types. For instance, the `Vector` class would have to be refactored to something like `Vector<owner>` in order to record the owner of each vector. Hence, while an implicit ownership type system is less expressive, the cost in complexity and the disruption to legacy code arguably outweighs the benefits of the added expressive power.

In addition, an important property of the static safety checks is that the restrictions they enforce only apply to the time-critical parts of the application code. In other words, the legacy code interacting with the Reflex task is by no means subject to the restrictions imposed by the static safety checks. One exception here is the data being shared between the time-oblivious code and the Reflex task; since such data is referenced from the Reflex task, it is required to be type safe. This scoping of the static safety checks is possible as the restricted unit is the Reflex task, and

not a thread as is the case with the `NoHeapRealtimeThread`. This clear distinction between Reflex code and time-oblivious code also has the effect of allowing for reuse of legacy code without or with only a few modifications and enables seamless integration hereof with Reflex graphs.

Rather than proving the assertion that a Reflex task that is valid according to the static safety checks is indeed type safe, the following sections informally describe a set of rules, that, together with the fact that a Reflex task is allocated in a separate private memory area, establishes by reasoning that neither the Reflex task nor the ordinary Java thread will experience dangling pointers nor observe heap-allocated objects in inconsistent states.

4.2 Partially Closed-World Assumption

A key requirement for type-checking a Reflex task is that all classes that will be used within it must be verified. To do so, we first construct a summary of classes, $\mathcal{W}$, used within a Reflex task based on an approximation of the live class set. The classes in $\mathcal{W}$ are categorized in three disjoint sets of classes: *stable*, *transient* and *capsule* classes.

Using classes to scope the static analysis represents a rather coarse-grained approach, it is simple and safe, and to a programmer it is intuitive – any class reachable from a Reflex task is subject to the type restrictions. Indeed, the approach is rather conservative in that it might include too many classes in $\mathcal{W}$. For instance, if a Reflex task (a class in $\mathcal{W}$) invokes a static method on some class C, simply returning a primitive type value, then C too is included in $\mathcal{W}$ and is subject to the restrictions regardless of any potential usage of the remainder of the class. Consequently, parts of the class C that would never be reached by the Reflex task could nevertheless cause for violations according to the restrictions.

The first thing the type checker has to ensure is that no class outside of $\mathcal{W}$ can be instantiated within any task in the program. This can be done in a straightforward fashion by inspecting the methods of the classes in $\mathcal{W}$ and checking that new objects are instances of class in $\mathcal{W}$.

> $\mathcal{R}1$: *Given a class instance creation expression* `new C(...)` *occurring in class* `C'`*, if* `C'` *or a subclass of* `C'` *is in* $\mathcal{W}$ *then* `C` *must be in* $\mathcal{W}$. □

The type checker will validate all classes in $\mathcal{W}$ and their parent classes. Classes that are not in $\mathcal{W}$ need not be type checked, effectively excluding from $\mathcal{W}$ the parts of the Reflex application that are not time-critical. The type checker will ensure that classes having static methods invoked from within a task belong in $\mathcal{W}$. Taken together rule $\mathcal{R}1$ and $\mathcal{R}2$ ensure that no object of a class that is not in $\mathcal{W}$ will ever be created while evaluating code in $\mathcal{W}$.

> $\mathcal{R}2$: *Any invocation of a* `static` *method* `C.m()` *occurring in class* `C'`*, if* `C'` *or a subclass of* `C'` *is in* $\mathcal{W}$ *then* `C` *must be in* $\mathcal{W}$. □

4.3 Implicit Ownership

The key property to be enforced is that all objects allocated within a Reflex task are encapsulated. This means that no object allocated outside of a task may refer to a stable or transient object of that task. Conversely, no stable or transient object may refer to an object allocated outside of the task. Since the Java application can hold a direct reference to a `ReflexTask` instance, a set of restrictive rules must be put in place to enforce this encapsulation.

> $\mathcal{R}3$: *The declaration of a non-private instance field of type* `T` *on a* `ReflexTask` *class, or a subclass hereof, is only allowed if* `T` *is a primitive type.* □

By only allowing reference fields in a `ReflexTask` class if they are declared `private`, we effectively prevent any reference fields from being assigned directly by the ordinary Java thread having a direct reference to the `ReflexTask` instance.

> $\mathcal{R}4$: *The declaration of a non-private method* `M` *on a* `ReflexTask` *class, or a subclass hereof, is only allowed if* `M` *is declared* `@atomic`. □

Prohibiting any public methods is actually a rather drastic solution. In fact, from the point of view of type safety, it would be sufficient to restrict any non-private methods from having reference type parameters or return types in their method signature. While this solution would be type safe, it would, however, provide a possibility to share data without proper synchronization. By enforcing that public methods are declared `@atomic`, we ensure synchronized access to shared data.

> $\mathcal{R}5$: *Methods declared* `@atomic` *on a* `ReflexTask` *class, or a subclass hereof, are restricted to declaring parameters of primitive and primitive array types, and can at most return primitive types.* □

Allowing the ordinary Java thread to invoke public transactional methods on the `ReflexTask` provides a potential risk of references leaking through the parameters of the method signature. Hence, method parameters are restricted to primitive and primitive array types, both having the property that they do not leak references. The return type is even more restrictive, permitting only primitive types. Allowing a reference to an object created during the invocation of the transactional method would be dangerous, as either it would be transient, and thus a dangling pointer once the ordinary Java thread exits the transactional method, or it would reference a stable object, potentially allowing references to leak this way.

A reasonable assessment to make concerning this restriction is whether it is too restrictive, and whether it would be possible to allow any reference type to be used as argument, rather than just

primitive arrays. Clearly, allowing all reference types to be used as arguments would not be safe, as this would allow for random references to leak in and out of the time-critical task. However, what about reference-immutable types in general? After all, primitive arrays are just a special case of reference-immutable type objects, which are characterized by not allowing references to mutate.

The nice property about primitive arrays is that it is statically determinable that they are reference-immutable. Allowing reference-immutable types in general as method arguments, one would similarly have to statically determine that the values that are actually passed are always of reference-immutable type. As we will describe in the next section, an exact set of possible value types is not computable statically, but must rely on an approximation. In other words, there would be cases where, given the lack of information available at compile time, it can not be statically guaranteed that providing reference-immutable types as arguments to transactional methods is always safe.

Furthermore, an additional requirement to these transactional method arguments would be to restrict them to transient types only. Passing in an argument to a heap-allocated stable object could cause for that object to be assigned by the ordinary Java thread to a stable field in the Reflex task. Type-wise the assignment would be valid, but unfortunately it would not be safe. As mentioned earlier, during the invocation of the transactional method, the reference type arguments are all pinned to ensure that they are not moved by the garbage collector. However, exiting the transactional method, these objects are unpinned again, and could from this point on be moved by the garbage collector. If at that point, a Reflex task held a reference to such a heap-allocated stable object that were subject to moving by the garbage collector, the task would observe the object in an inconsistent state or see a dangling pointer.

Finally, like is the case for the primitive array types, general reference-immutable types would have to be pinned when used in the transactional method. However, anything referred to from the reference-immutable type would have to be pinned too. Consequently, using reference-immutable types as method arguments would require a recursive pinning and unpinning upon every transactional method invocation, a process that is uncontrolled and can be a potentially variable source of delay.

Given these circumstances, in the current version of Reflexes we refrain from allowing reference type arguments to transactional methods (besides primitive array types), noting at the same time, that transactional methods can freely use variables reference-immutable types if they are declared static. Why this is possible, we will describe in a bit.

Another possible approach would be to allow exchange of objects by deep copy. This is safe and would benefit from the fact that the same Java type can be used for both transient/stable objects within a Reflex and for normal objects allocated on the heap. The downside, however, is the overhead introduced by performing this deep copy, which, e.g., even for primitive arrays can be significant if they have arbitrary length.

Finally, the reflective instantiation of a `ReflexTask` prevents any references from leaking in during task instantiation since only default constructors are invoked by the Reflex run-time engine, preventing reference types to leak in. Any attempt to circumvent this reflective instantiation

will result in an object that is not a Reflex since its private memory area and all its internal structures are lacking.

> $\mathcal{R}6$: *Instantiating a* `ReflexTask` *class only results in a Reflex when that instantiation is done reflectively by the Reflex run-time engine.* □

Dangling pointers within the `ReflexTask` instance are prevented by segregating stable from transient references. No (long-living) stable object may acquire a reference to a (short-lived) transient object. This is done at the class granularity. If a class is declared stable, then it can only refer to other stable classes.

> $\mathcal{R}7$: *The type* `T` *of an instance field declaration in a stable class or a parent of a stable class is legal if* `T` *is a primitive type or if* `T` *is a stable class.* □

Since the set of static safety checks tracks classes, it is critical to prevent instances of transient classes from masquerading as stable types. This is achieved by mandating that descendant of stable classes are stable.

> $\mathcal{R}8$: *Assume* `C` *is a stable class in* $\mathcal{W}$*, for any class C' in* $\mathcal{W}$*, if* `C' extends C` *then* `C'` *must be stable.* □

Following from here, since the `ReflexTask` class is declared stable by implementing the `Stable` interface, any subclass hereof can only declare instance fields of primitive or stable types.

It should be noted that the above rule does not prevent a class declared stable in some $\mathcal{W}$ to have subclasses that are not stable. That is allowed as long as these are not in $\mathcal{W}$, i.e., not used from within a Reflex task. Nothing prevents an ordinary Java thread from allocating a stable class. This poses no problem as it will be allocated using standard Java semantics in the memory context of the public heap where the stable property is harmless beyond of course indicating that it is a type that (potentially) could be assigned to a stable field in a Reflex task. However, the static safety checks described so far would prevent this reference from leaking in to the `ReflexTask` instance.

4.4 Static Reference Isolation

Enforcing encapsulation also requires that communication through `static` variables be controlled. Without any limitations, static variables can be used for unrestricted sharing of references across encapsulation boundaries and thus open up opportunities for all kinds of programming errors.

A drastic solution is simply to prevent code in $\mathcal{W}$ from reading or writing static reference variables. Clearly this is safe as the only static variables that a task is allowed to use are ones with primitive types. The question is of course how restrictive is this rule? While it may be the case that for newly written code it is straightforward to replace static variables with context objects that are threaded through constructors, the same can not be said for library classes as it would be difficult to refactor them and if one did, they would loose backwards compatibility. Having too restrictive rules may thus seriously hamper code reuse.

The key observation to circumventing this restriction is that static variables are not dangerous if they are not referenced or are never modified. This suggests introducing the notion of *reference-immutable* types. These are types that are transitively immutable in their reference fields and possibly mutable in their primitive fields.

> $\mathcal{R}$9: *A field* `F` *in class* `C` *is effectively final if it is either (1) declared* `final` *and of reference-immutable type, or (2) declared* `private`*, of reference-immutable type, and not assigned in any non-constructor methods in class* `C` *and parent classes of* `C`*.* □

> $\mathcal{R}$10: *A class* `C` *in* $\mathcal{W}$ *is reference-immutable if all non-primitive fields in the class and parent classes are effectively final.* □

The analysis can infer which types must be immutable based on the use of static variables. With the introduction of *reference-immutable* types, statics can restrictively be used without compromising encapsulation.

> $\mathcal{R}$11: *Let* `T` *be a class in* $\mathcal{W}$ *or a parent of a class in* $\mathcal{W}$*. A* `static` *field access expression occurring in* `T` *is legal if the field is a primitive or if the field is effectively final and it can be statically determined that it is assigned a null or a value of reference-immutable type. An assignment statement occurring in* `T` *is legal if the left-hand side of the assignment is a* `static` *field of a primitive type.* □

This last rule represents a pragmatic choice balancing the desire for expressiveness in order to reuse a maximal amount of pre-existing library code and the ability to statically ensure type-safety.

However, it turns out that enforcing this rule statically is non-trivial. In fact, this boils down to the problem of inferring the exact set of live classes, as mentioned earlier. The problem of the rule centers around how to *statically determine* which values are assigned to static fields that are read by Reflex tasks. In fact, because of the possibility for subtyping, it is not sufficient to look at the type of the declared field only, but also the possible types of the values that can be assigned to the field. For instance, say that the non-final type of the declared static field is T. If T' is a subclass of T, then the field may be assigned a value of type T' (or a subclass hereof),

in addition to a value of type T. However, while T might be a reference-immutable type, T' is not necessarily as it might declare mutable reference type fields. Yet both values of type T and T' can be assigned to the field.

Contrary, if the type of the declared field is a final type (declared `final`) and is a reference-immutable type according to $\mathcal{R}10$, then it follows that reading from this static field is safe – only a `null` value or a value of reference-immutable type can be read. However, for all non-final types, for a static field assignment to be safe, it needs to be statically determined that only a null or a value of reference-immutable type can be assigned to the static field.

Ensuring the set of live classes with exact precision is not possible. Instead, we use an approximation based on the following principles. This set of live types $\mathcal{W}$ for the static variable can be found by analyzing the class initializer, or `<clinit>`, of the class declaring it, and from here looking at all the types that are used directly or indirectly by the class initializer. To calculate this live set, we use a simple and conservative algorithm where all methods reachable from the class initializer are analyzed together with their bytecodes, thereby refraining from doing any control-flow and data-flow analysis. The algorithm is shown in Fig. 4.2.

```
function analyzeMethod(M)
  set live set S_live := {∅}
  for all instructions i in method M do:
    if (i is a new instruction of type T) then
       S_live := S_live ∪ {T}
    else if (i is an invocation of a method M' with return type T) then
       if (T is void) then
         ignore, does not affect live set
       if (T is declared final) then
         S_live := S_live ∪ {T}
       if (T is not declared final) then
         potential safety problem!
    else if (i reads a static variable declared in type T) then
         S_live := S_live ∪ {analyzeMethod(<clinit> method of T)}
    else if (i reads an instance variable of type T)
       if (T is declared final) then
         S_live := S_live ∪ {T}
       if (T is not declared final) then
         potential safety problem!
```

Figure 4.2: A simple and conservative algorithm for analyzing the live set of classes, S_{live}, in the class initializer, `<clinit>`. Specifically, this algorithm is used to infer the possible types that can be assigned to a static variable having a reference-immutable type.

Having analyzed the class initializers and calculated a live set of classes from here, all classes that are type incompatible with the type of the static field being read are discarded from the live set as they can never be assigned to the field. The remaining classes in the live set then are checked for reference-immutability following the rule in $\mathcal{R}10$, and in the event that one or more

types are not reference-immutable, there is a safety problem, and the violating code statement will have to be rejected.

The ability to use static variables of reference-immutable type from within a Reflex task raises an interesting question; does a static variable of reference-immutable type, T, pose a safety problem if T is also declared stable? The potential problem relates to the fact that a heap-allocated object hereby could be stored in a stable object within the task's private memory area. From rule $\mathcal{R}7$ we know that an object of stable type may point to any object of stable type. Consequently, since the `ReflexTask` class is inherently declared stable, it may declare stable instance fields and assign references to the heap-allocated stable object. The potential dangers here are twofold; first, could references thereby leak in or out, and second could a dangling pointer occur since the reference crosses memory boundaries? The former threat is eliminated by the fact that the type T is required to be reference-immutable, according to rule $\mathcal{R}11$. The latter threat can be handled with some help from the virtual machine. In fact, the Reflex programming model relies on the virtual machine to provide functionality to pin objects to their location on the public heap. More specifically, the virtual machine will pin any static, heap-allocated variable (stable or not) referenced by the `ReflexTask` instance throughout the lifetime of the `ReflexTask` instance, preventing the garbage collector for moving and removing the variable. This way the `ReflexTask` instance can safely reference the heap-allocated reference-immutable static variable, as specifically illustrated in Fig. 4.1. Moreover, since any transient object can reference a stable object, the heap-allocated static variable can also safely be used from the context of a transient area of the `ReflexTask` instance.

Note also, rule $\mathcal{R}11$ prevents any stable class (that would be in $\mathcal{W}$) from declaring static variables of reference-immutable type as these would necessarily have to be assigned in the class initializers of the stable class, and $\mathcal{R}11$ prevents any assignment to a static variable.

Finally, we assume that all static variables are initialized a priori to the instantiations of the Reflex tasks in the graph. More specifically, all class initializers of classes containing static variables accessed by Reflex tasks must have been invoked before the graph executes. Without this assumption, static variables could potentially be assigned a value allocated in the transient area of the Reflex task causing for the class initializers to be invoked, which would lead to a dangling pointer.

At this point, an interesting question is why allow static variables of reference-immutable type, but not transactional method arguments of reference-immutable type? There are really three reasons: First, to use reference-immutable types as transactional method arguments not only is an approximation of the set of possible reference-immutable types required. Since transactional method arguments are restricted to transient types only would have to make an additional approximation over which of the possible reference-immutable types are also transient. Second, the continuous pinning and unpinning of an arbitrarily large reference-immutable type would have to be further investigated in order to determine its impact on performance. Third, although the reliance on static variables is often seen as poor programming practice, by combining transactional methods with the use of static variables of reference-immutable type, one can achieve more or less the same functionality as provided when passing in reference-immutable types to the transactional methods. Thus, to summarize, we leave transactional method arguments of

reference-immutable type as an opportunity to be exploited as future work.

4.5 Capsules

A capsule is an object that is manipulated in a linear fashion. At any given time, it must be enforced that the following holds: (1) there is at most a single reference to the capsule from a single data channel, and (2) there can be multiple reference on the stack and in fields of transient objects. With these invariants the implementation can achieve zero copy management of capsules.

> $\mathcal{R}12$: *A capsule is an instance of a subclass of* `Capsule`*, which (1) declares only fields of primitive types and* `final` *primitive array types, and (2) declares only* `private` *constructors.* □

The above rule is a pragmatic choice that effectively and easily ensures that capsules are reference-immutable (without permitting general reference-immutable data structures) and can only be instantiated by the Reflex run-time engine. The motivation is that the Reflex run-time engine must be in charge of allocating and reclaiming capsules. Contrary, it would be possible to allocate a capsule in transient memory and push to an output channel, eventually leading to dangling pointer error.

> $\mathcal{R}13$: *Capsule types in $\mathcal{W}$ are treated as transient types and are consequently not allowed to implement the* `Stable` *marker interface.* □

From the point of view of stable and transient classes, a capsule is "just" like any other transient class. Thus, we inherit the guarantee that when `execute` method returns there will be no reference to the capsule in the state of a Reflex task.

4.6 Arrays

As a consequence of the default allocation context being transient, any primitive array type is considered a transient type. With this categorization, Reflex tasks cannot declare instance fields of primitive array types, as mentioned earlier, following $\mathcal{R}7$. Instead, `ReflexTask` s must use the set of array wrappers for encapsulating primitive arrays in the stable heap, as described in Sec. 3.3.5. Through this distinction, primitive arrays can be used safely in both memory contexts.

Besides being able to create arrays of primitive types through the use of the `StableArray` classes, nothing, however, prevents a Reflex task from also creating arrays of stable type. In fact, if a

class is stable then the array class derived from that class is stable too. Thus, expressions like `Object[] ms = new MyStable[10]` are perfectly valid since the array of `MyStable` itself is a stable object, and transient objects, like `ms`, are allowed to reference stable ones. Of course, even though `ms` can reference an array of stable type, its own declaration `Object[] ms` is still transient (since `Object` is transient), and thus a field declaration of type `Object[]` in a stable class would not be valid, according to $\mathcal{R}7$.

4.7 Other Restrictions

In addition to the above mentioned rules, the static safety checks impose the following restrictions:

Finalization Every Java class inherits the `finalize` method from `java.lang.Object` class. This method is normally invoked by the garbage collector when no other references are pointing to the object, giving the programmer an opportunity to perform manual cleanup of typically external resources before the object is being irrevocably discarded.

In Reflexes, the virtual machine does not invoke finalizers of transient objects allocated in a `ReflexTask`; allowing finalizers would violate the constant time deallocation guarantee of the transient area. Furthermore, given the isolated nature of the Reflex tasks, and their short-lived nature, the justification and necessity for finalizers is doubtful.

Thread Creation A Reflex task is not allowed to create and spawn any threads using the `java.lang.Thread` class, or any of its subclasses, as doing so would would affect scheduling.

Specialized References Any of the `java.lang.ref.Reference` types (weak, soft, and phantom references) are not permitted within a Reflex task; semantically, they do not make sense in the context of the transient area as all such transient allocations are not garbage collected anyway.

Native Code While not restricted by the static safety checks, native code invoked from Java methods poses a specific problem. Particularly, since the invocation will exit the virtual machine, it is basically out of the control of the Reflex run-time engine, and could here perform various operations that could have a negative impact on predictability in addition to being type unsafe, e.g., through JNI callbacks. Rather than automatically restricting such methods, the Reflex programming model relies on the programmer to manually validate the impact of the native methods invoked, in other words, the programmer must ensure not to invoke native code that would violate type safety or impact predictability.

4.8 Class Library Reuse

An interesting question is, how restrictive the proposed set of static safety checks really is, and to what extent it allows for reuse of the standard Java class library? As an experiment, we tried to type-check the collection classes, such as `Vector` and `HashMap` in the `java.util` package for Java 1.4 (the GNUClasspath open source implementation). When inner classes are counted, there are 126 classes and about 22,000 lines of code. We decided to treat the collection classes as transient and ran the type checker.

The first set of errors were due to the use of classes such as `String`, `StringBuffer` and `Random`. We declared them as intrinsics – special types that are treated as transient by the type checker but considered safe, and thus not validated. After declaring these classes safe, there were still over 200 type errors due to the use of static reference variables. The collection classes have a total of 66 static fields, out of which only 10 fields are of reference type. They hold various singletons used to represent empty collections, empty slots and empty keys. The key observation is that these statics are examples of such that are never modified and are never reclaimed. Adding rules allowing for reference-immutable static types eliminated all but a handful of errors.

Rooting out the last errors would require some refactoring of the collection classes. The problem arises from the fact that some of the singletons, while they are in practice immutable, have non-`final` fields. One can take care of those errors by refactoring some of the collection classes to introduce immutable singleton classes. There is only one class that we did not attempt to include in the experiment, `WeakHashMap`, as it drags in extra libraries and has no use within a Reflex task since transient objects are not garbage collected.

In conclusion, we found that the majority of Java collection classes can be used without changes within a Reflex task and a small number of classes require small modifications to be usable.

Part III

Implementation

5 Reflex Implementation

Having described the Reflex programming model and the set of static safety checks enforced on the Reflex applications, this chapter describes our initial prototype implementation of Reflexes on a research real-time Java virtual machine, highlighting the most interesting aspects of the implementation.

5.1 Implementation Overview

As implementation platform, Reflexes uses the Ovm [BCF+06] real-time Java virtual machine, which comes with an optimizing ahead-of-time compiler and provides an implementation of the Real-time Specification for Java (RTSJ). The virtual machine was designed for resource constrained uni-processor embedded devices and has been successfully deployed on a ScanEagle Unmanned Aerial Vehicle in collaboration with the Boeing Company [ABC+06].

We leverage the RTSJ support in Ovm to implement some of the key features of the Reflex API. For instance, the stable and transient memory areas in a Reflex task are implemented using standard RTSJ `ScopedMemory`, and the threads executing the Reflex tasks are subclasses of the standard `RealtimeThread` construct. The virtual machine configuration described here uses an optimizing ahead-of-time compiler to achieve performance competitive to commercial virtual machines [PV06]. Furthermore, in our implementation, we switched off any memory boundary checks on the `ScopedMemory` that are normally performed by RTSJ-compliant virtual machines, as we provide these guarantees statically through our static safety checks.

5.2 Scheduling

Scheduling is implemented in the virtual machine. Ovm supports a scheme based on priority-preemptive scheduling for real-time threads with a complete range of priorities from 1-42, the subrange 12-39 are real-time priorities used by Reflex tasks and the remaining are used for ordinary Java threads. The virtual machine's mostly-copying garbage collector is run in an ordinary Java thread.

The `ReflexTask` instances in each Reflex graph are executed by a single thread with real-time priorities according to the priority given to the graph it belongs to. The thread is started as a result of an invocation of `start` on the Reflex graph, which basically causes the thread of the `Clock` task to start. Having started, upon reaching its period, the `Clock` will invoke put the latest time stamp on channel, and traverse downstream in the graph and execute any tasks that are schedulable, as illustrated by the algorithm in Fig. 5.1.

```
upon next period arrived
  for all outgoing time channels C_out of clock task do
    clock task pushes timestamp on C_out using C_out.putTime();
    inspectDownstreamTask(C_out.targetTask);

function inspectDownstreamTask(T)
  for all incoming channels C_in of task T do
    if (elements in C_in >= rate of C_in) then // if size of inbound channel is equal or larger than expected rate
      T.execute();
      for all outgoing channels C_out of task T do
        inspectDownstreamTask(C_out.targetTask);
```

Figure 5.1: An algorithm showing how tasks in a Reflex graph are executed in the Reflex prototype implementation. The clock task triggers the execution by pushing a timestamp on its outgoing channels after which the thread traverses downstream in the graph, executing any receiving task that is schedulable.

With this single-threaded scheme, special care must be taken by the programmer to ensure that the worst case graph traversal time is less than the period selected in the clock. If this is not the case, inevitably the execution will span multiple periods, and deadline misses will occur.

5.3 Memory Management

For each `ReflexTask`, the implementation allocates a fixed size continuous memory region for the Reflex's stable area and another region for its transient area. The size of each of the above is set programmatically in the Reflex API, as shown in Fig. 3.3. Furthermore, a buffer is set aside for the transactional log. In our prototype implementation, the size of the transaction log is growable, but not shrinkable, but the log can be reset and already allocated entries reused

between transactions. In other words, the size of the transaction log will at any point correlate with the maximum number of memory mutations made in an invocation of a transactional method. However, recall, that the transaction log only holds mutations to stable objects. The `ReflexTask` object, the transaction log and all other implementation specific data structures are allocated in the Reflex task's stable area, and thus not subject to garbage collection.

Each thread in the virtual machine has a default allocation area. For ordinary Java threads, this area is of course the public heap. For real-time threads executing the Reflex task's `execute` method, this area is the transient area of the task. Likewise, when an ordinary Java thread invokes a transactional method on a Reflex task, the memory area has to be switched to the transient area of the task throughout the invocation, and reset once the invocation returns. To enable this, we have modified the bytecode rewriter of the Ovm compiler to bracket all invocations of transactional methods declared on the `ReflexTask` subclass with `setCurrentArea/reclaimArea` invocation pairs to ensure that when a transactional method is called the allocation area is set to the transient region, as seen in Fig. 5.2.

```
@atomic public void update(int[] ia) {
  MemoryArea publicHeapArea = ReflexSupport.setCurrentArea(this.transientArea);
  try {

    // body of method goes here
  }
  finally {
    ReflexSupport.reclaimArea(this.transientArea);
    ReflexSupport.setCurrentArea(publicHeapArea);
  }
}
```

Figure 5.2: An illustration of the effects of the modification made to the bytecode rewriter of the Ovm compiler, wrapping the method body of a transactional method with invocations to switch and reclaim the allocation context of the invoking thread. Note, we use Java source code to illustrate the effects of the modification, but in fact they are performed directly on the bytecodes.

As can be seen from Fig. 5.2, the virtual machine exposes low-level functionality for setting allocation areas through the `ReflexSupport` class. The method `setCurrentArea` allows the Reflex implementation to change the allocation area for the current thread. Regions are reference counted, each call to `setCurrentArea` increase the count of active threads by one. The method `reclaimArea` decreases the counter by one for that area, if the counter is zero all objects in the area are reclaimed. Essentially, the allocation pointer is reset to the start of the area.

Besides switching memory context, the virtual machine also is responsible for redirecting the allocation of stable classes into the stable heap. For this purpose, the virtual machine exposes another native method, `setAllocKind(graph, class)` for internally identifying stable classes. This method is invoked by the Reflex run-time once for each stable class used by the tasks in the Reflex graph to be executed. The list of stable classes is provided to the Reflex run-time engine

as a result of the type checking of the given Reflex graph, as detailed later. By invoking this method for a given class T, whenever an instance of T is created within the particular Reflex graph, the stable heap of the current task is used as allocation context for the object.

The virtual machine also supports allocation policies for meta-data. In particular, we rely on a policy for lock inflation that ensures that a lock is always allocated in the same area as the object with which it is associated, regardless of the current allocation area.

5.4 Transactional Methods

To implement the transactional methods, we exploit the *preemptible atomic regions* [MBC+05] facility of the Ovm virtual machine, a non-standard facility not supported by standard compliant commercial Java virtual machines.

By exploiting the preemptible atomic regions facility any method annotated `@atomic` will be treated specially by the Ovm compiler during compilation. More specifically, the compiler will privatize the call-graph of a transactional method, i.e., recursively generate a transactional variant of each method reachable from the transactional method. This transactionalized variant of the call-graph is invoked by the ordinary Java thread, whereas the non-transactional variant is kept around as the Reflex task might itself invoke (via the `execute` method) some of the methods, and those should not be invoked with transactional semantics as they will always succeed and thus never roll-back. The transactional method itself is not privatized; only any methods that are invoked from here. In our prototype implementation, we do not prevent the `execute` method from invoking a transactional method and thus the transactional variant of the call-graph. However, the effects would not be dangerous; we know that the `execute` method will always complete. Indeed, the Reflex thread invoking the `execute` method can be preempted by a higher priority Reflex thread executing another task (in another Reflex graph), but it will eventually complete and never roll back. In this light, the fact that part of the invocation is executed with transactional semantics poses no problem, though we discourage it.

Furthermore, we have applied a subtle modification to the preemptible atomic region implementation. Rather than having a single global transaction log, a transactional log is created per `ReflexTask` instance in the graph, assuming that it declares transactional methods. This change ensures the encapsulation of each `ReflexTask` instance, and enables concurrent invocation of different transactional methods on different `ReflexTask` instances.

The preemptible atomic regions use a roll-back approach in which for each field write performed by an ordinary Java thread on a stable object within the transactional method, the virtual machine inserts an entry in the transaction log and records the original value and address of field. With this approach, a transaction abort boils down to replaying the entries in the transaction log in reverse order. Running on a uni-processor virtual machine, no conflict detection is needed. Rather, the transaction aborts are simply performed eagerly at context switches. Specifically, the transaction log is rolled back by the high-priority thread before it invokes the `execute` method of the schedulable Reflex task. Whereas the complexity of transaction aborts is proportional with

the number of writes performed in the transactional method at the time of preemption, starting and committing a transaction can be done in constant time. Upon resuming, the ordinary Java thread will discover that it was preempted, and will subsequently retry the invocation of the transactional method.

Should an exception occur within the transient area of a Reflex task, i.e., during the invocation of the `execute` method, we rely on standard Java semantics that will cause the exception object and its stack trace to be allocated in transient memory. If the exception propagates out of the `execute` method, the stack trace is printed and the Reflex task computation is terminated. Contrary, if the exception occurs during an ordinary Java thread's invocation of a transactional method and the exception propagates out of the outermost transactional method, we rely on standard RTSJ behavior. The problem here is that the exception object is allocated in the transient area within the task, and thus is out of reach of the receiving Java thread allocated on the public heap. Leveraging RTSJ specific behavior, rather than receiving the specific exception object, the ordinary Java thread will receive an unchecked `ThrowBoundaryError` with a `String` based description of the actual thrown exception.

5.5 Pinning of Objects

As mentioned earlier, for type safety reasons static variables of reference-immutable type referenced from within a Reflex task are required not to be moved by the garbage collector in order to prevent a dangling pointer or observing the object in an inconsistent state. For static variables, our implementation exploits the fact that standard RTSJ allocation policy for classes and static initializers ensures that all objects are allocated in the `ImmortalMemory` area, a non-garbage collected region.

The Ovm garbage collector supports pinning for objects. Pinned objects are guaranteed not to move or be removed during a garbage collection, and will therefore always be in a consistent state when observed by referent objects from other memory areas. Thus, they can safely be accessed from a Reflex task. Although, we do not pin static variables, there is one other case where pinning is required. Arguments to transactional methods are heap-allocated objects and are thus pinned as the ordinary Java thread invokes a transactional method on the Reflex task and unpinned again when the invocation exits. We have modified the bytecode rewriter of the Ovm compiler to instrument the method bodies of the outermost transactional methods (those methods on the `ReflexTask` declared `@atomic`) to pin and unpin any reference type objects passed in. Fig. 5.3 illustrates in Java source code the effect of the modification of the Ovm compiler, more specifically, the insertions of the invocations to the `pinObjects` and `unpinObjects` methods.

```
@atomic public void update(int[] ia) {
  ReflexSupport.pinObjects(new Object[]{ ia });
  try {

    // body of method goes here
  }
  finally {
    ReflexSupport.unpinObjects(new Object[]{ ia });
  }
}
```

Figure 5.3: An illustration of the effects of the modification made to the bytecode rewriter of the Ovm compiler, inserting methods to pin and unpin reference type arguments provided to the outermost transactional methods. Note again, we use Java source code to illustrate the effects of the modification, but in fact they are performed directly on the bytecodes.

5.6 Static Type Checker

The Reflex type checker is implemented as a pluggable type system through an extension of the JavaCOP framework [ANMM06]. JavaCOP is a system for defining and enforcing constraints on the structure of a Java program. It consists of a rule language for specifying semantic and structural constraints on Java programs as well as a framework for validating structural and semantic constraints. In our case, we have refrained from specifying the Reflex static safety checks using the JavaCOP language, and instead simply extended the JavaCOP framework and hardcoded the safety checks directly. The Reflex type checker is implemented as an extension of the `javac` 1.5 compiler, adding an extra pass in the `javac` 1.5 compiler. This approach is convenient as the rules are fairly compact and that error messages are returned by the Java compiler – no extra tool is required and messages are returned with line numbers in a format that is familiar to programmers.

Specifically, the Reflex static type checks is implemented by extending the `AbstractConstraints` class of the JavaCOP framework and contains around 900 lines of code. This class implements a *visitor* pattern [GHJV95], and can intercept and handle the visit of each node in the abstract syntax tree of the Java program, where each node represents parts of the programs structure: classes, methods, blocks, statements, expressions, identifiers etc.

Upon starting up, the Reflex type checker reads a text file containing the intrinsic types, i.e., those that have been manually declared safe types by the programmer, before commencing on the actual checks. To this initial set belongs all the base classes of the Reflex API. The checks are performed purely on a class-based level, which for the stable/transient checks is necessary anyway. However, for the checking of reference-immutability, as described in the algorithm in Fig. 4.2, the class-based level has its limitations as the actual values assigned to a field are not know statically. Thus, should the Reflex type checker not be able to statically determine that a given type assigned to a static variable is indeed reference-immutable, the checker will

report an error to the user, and thus the checking will fail. Due to the imprecision of the approximation, this might or might not represent a real error. If it turns out not to be, and the programmer is certain that the type is indeed reference-immutable and thus safe to assign to a static variable, the type can be added to the file containing the intrinsic types, whereby its usage will no longer cause an error to occur during type checking.

Upon a successful safety checking, the Reflex type checker generates an output file containing the list of all stable types that were found in the checked application and used from within the Reflex tasks. These types are found from a starting point consisting of classes implementing the `Stable` marker interface and augmenting this set with those classes found according to the static safety checks $\mathcal{R}7$ and $\mathcal{R}8$. The output file is subsequently read and used by the Reflex run-time engine upon executing the checked application, and informs the Reflex run-time engine about which types should be allocated in the stable heaps of the tasks, as described earlier.

6

Empirical Prototype Evaluation

This chapter reports on a number of empirical evaluations of our prototype implementation of Reflexes when executed on a research virtual machine. We evaluate Reflexes on two dimensions; predictability and performance, both important properties of real-time systems. For this purpose, we employ two benchmark applications and compare these to equivalent variants written in C. We also report on a benchmark comparison of our research virtual machine with others, to take into account their performance profiles.

6.1 Methodology

The purpose of the evaluation is to expose to which extent applications programmed with Reflexes achieve the response times of highly responsive applications, and how one such application compares to an equivalent C variant in terms of performance.

All the conducted experiments were performed using a variant of the Ovm virtual built with support for POSIX high resolution timers, and configured it with an interrupt rate of 1 μs, disabled the run-time checks of violations of memory region integrity (read/write barriers) and configured the heap size to 512MB. Finally, non-determinism due to just-in-time (JIT) compilation is avoided through the ahead-of-time compilation provided by Ovm.

As execution platform we used an AMD Athlon 64 X2 Dual Core processor 4400+ with 2GB of physical memory running Linux. The kernel version used was 2.6.17 extended with high resolution timer (HRT) patches [RT] configured with a tick period of 1 μs.

We evaluate Reflexes based on two metrics of predictability: precision of inter-arrival time, i.e., time between two successive executions of a Reflex task, and number of missed deadlines when

running Reflexes in isolation and in a mixed environment.

At this point, it is important to properly characterize a deadline miss. A missed deadline occurs for the i'th firing of a Reflex with a period p if the actual completion time, α_i, comes after its expected completion time, ϵ_i, where $\epsilon_i = p(\lceil(\alpha_{i-1}/p)\rceil + 1)$. When counting missed deadlines, we will be conservative and consider both *real* deadline misses as well as *absolute* deadline misses; the difference being that for a given period p, a single absolute deadline miss might cover $i * p$ real deadlines, where $i > 1$. (So we will in this case count i deadline misses.)

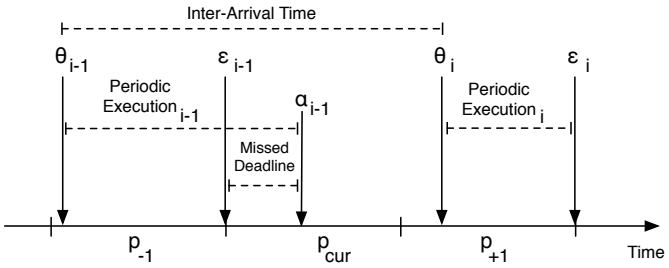


Figure 6.1: Timeline showing how a missed deadline can cause an inter-arrival time between two consecutive periodic executions to be larger than twice the period.

Note also that an inter-arrival time larger than twice the period p (but strictly less than three times the period) does not necessarily imply more than a single deadline miss. Fig. 6.1 shows that in the event of a deadline miss (when actual completion time, α_{i-1}, lies after the expected completion time, ϵ_{i-1}) of a $i-1$'th firing, the expected completion time, ϵ_i, of the subsequent i'th firing is set to be the end of the first-coming complete period, i.e., any time remaining in the current period is skipped. If the start of the subsequent periodic execution, i, is delayed (reflected in the actual start time, θ_i, lying after the period start) it can cause the inter-arrival time between the two consecutive periodic executions, $i-1$ and i, to be larger than twice the period p.

6.2 Virtual Machine Benchmarks

Working on a research virtual machine always raises questions about applicability of the results in the context of 'real' systems. We report on the raw performance of Ovm on the SpecJVM98 benchmark suite and compare it with Hotspot 1.5 and GCJ 4.0.2. We evaluate two Ovm configurations: the plain Java configuration and the RTSJ configuration which includes scoped memory access checks. Fig. 6.2 shows that Ovm outperforms GCJ and fares surprisingly well when compared to a commercial virtual machine.

The figure also illustrates the costs of RTSJ read/write barriers (up to 50%). SpecJVM is by no means representative of a real-time application, but it gives an estimate of the cost of memory access checks.

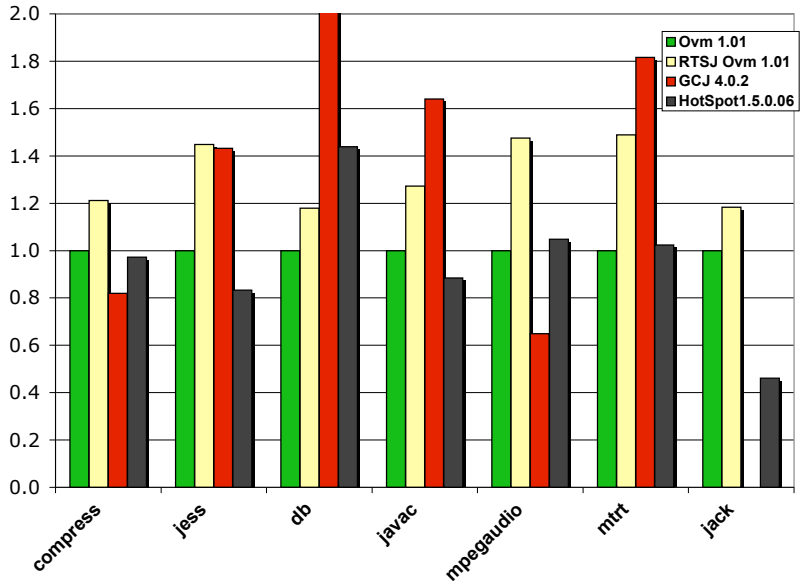


Figure 6.2: Comparing Java VMs on the SPECJVM98 benchmarks. The x-axis shows the individual benchmark tests and the y-axis the relative performance compared to Ovm (set to 1.0).

6.3 Evaluation: Single Task Graphs

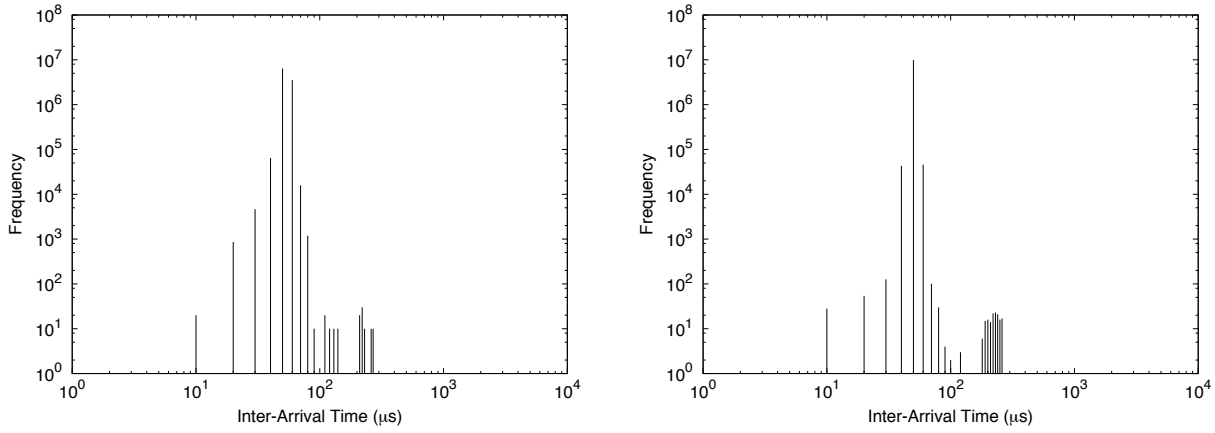
First, we considered benchmark applications consisting of single-task Reflex graphs. More specifically, we experimented with a null task, i.e., a task with an empty `execute` method, to get some baseline measurements on predictability, after which we adopted a music synthesizer to measure performance using a standard audio periodicity.

6.3.1 Predictability

To evaluate predictability of Reflexes, we implemented a simple Reflex graph containing a single null task, scheduled it for a 45 μs period (equivalent to frequency of 22.05 KHz, a standard audio frequency), and let it execute over 10 million periods. We also implemented a C variant of the same code, though the C variant relies on POSIX real-time extensions.

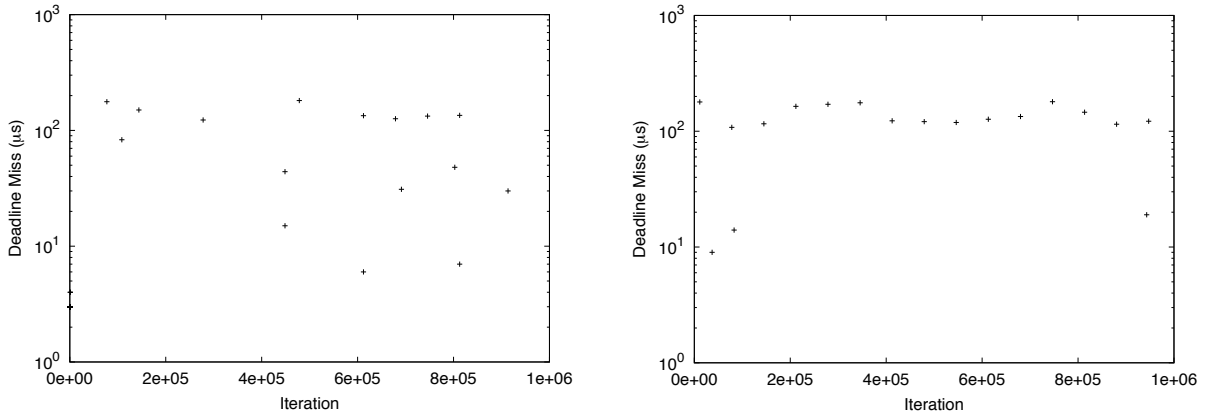
As depicted in Fig. 6.3 nearly all interesting observations centered around the 45 μs period, though the Reflexes appear to be slightly less timely than the C variant, because the spread in inter-arrival time is wider. Also note the observations clustered around 200-250 μs for both variants, which we attribute to perturbations in the underlying operating system. Similar observations for an equivalent base performance benchmark are reported in [SAB+06].

Fig. 6.4 depicts missed deadlines for both Reflexes and the C variant. More precisely, with Reflexes 99.996% of the periods are completed in time with no absolute deadline miss (99.993% in the case of real deadline misses). On the other hand, the equivalent for the C variant is 99.997%



(a) (b)

Figure 6.3: Histograms of inter-arrival time for (a) Reflex graph with a null task scheduled for 45 μs periods, and (b) an equivalent the C variant also scheduled for 45 μs periods. The x-axis shows the logarithm of the inter-arrival time in μs and the y-axis shows the logarithm of its frequency.



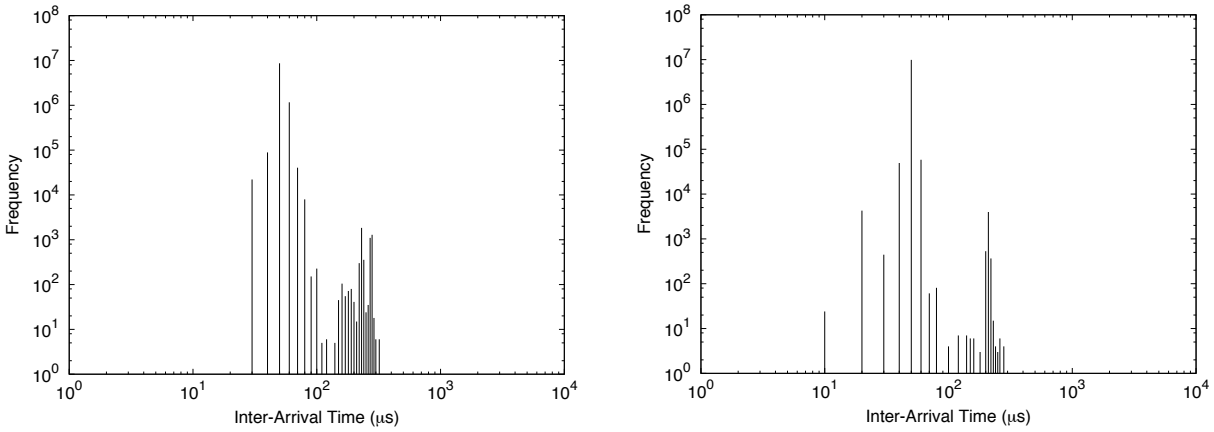
(a) (b)

Figure 6.4: Missed deadlines over time for respectively (a) Reflex graph with a null task scheduled for 45 μs periods, and (b) an equivalent the C variant also scheduled for 45 μs periods. The x-axis shows the executions (only 1 million iterations shown) of the periodic task and the y-axis shows the logarithm of the size of the deadline misses.

(real: 99.993%). Interestingly, Fig. 6.4 indicates some pattern in deadline misses around 100-200 μs for both Reflexes and the C variant, though for the C variant there seems to more consistency in that pattern. Also, it appears that both versions experience an equivalent amount of deadline misses, but Reflexes have more variation in the actual sizes of the misses than the C variant. In both cases, given the similar patterns in the missed deadlines lead us to believe that these must be caused by the underlying operating system.

6.3.2 Performance

Having compared the predictability of Reflexes and corresponding C code, we next measured the performance of Reflexes under a workload stemming from a mixed environment with an ordinary Java thread and a (time-critical) Reflex task executed concurrently. We considered here a music synthesizer application, developed at IBM Research for Eventrons [SAB+06], which we modified to make use of Reflexes, including a transactional method.[1] In short, the scenario involves an ordinary Java thread that generates music samples, and writes these to a buffer on the `ReflexTask` instance through a transactional method. These samples are then periodically read by an audio player Reflex scheduled with 45 μs periods and which then writes the samples individually to the sound device for playing. Note, this is an example of an application where we successfully use native methods though, as mentioned earlier, in general their effects on time-critical tasks might make them unsuitable. For the sake of comparison, we implemented a corresponding C variant of the music synthesizer.



(a) (b)

Figure 6.5: Histograms of inter-arrival time for respectively a (a) Reflex and (b) C variant of an audio player task scheduled for 45 μs periods. The x-axis shows the inter-arrival time in μs and the y-axis shows the logarithm of its frequency.

Fig. 6.5 depicts the inter-arrival time of the time-critical audio player thread for both the Reflex and C variants. As already noted in Fig. 6.3, outlier clusters around the 200-300 μs range can also be seen in Fig. 6.5 for both Reflexes and its C variant. However, in Fig. 6.5 these outliers appear to have been enhanced, which we attribute to the effects of buffering congestion in the sound device to which the time-critical task is writing (twice per execution)[2].

The outlier clusters seen in Fig. 6.5 also seem to have a direct impact on the missed deadlines as seen in Fig. 6.6. Specifically, for Reflexes 99.869% (real: 99.698%) of them complete in time and do not cause deadline misses. For the C variant, this is the case in 99.949% (real: 99.799%) of

[1] Eventrons have been released under the name *Expedited Real-Time Threads* and is available from `http://www.alphaworks.ibm.com`.

[2] First the upper 8 most significant bits of the short value are written to the sound device followed by the 8 least significant.

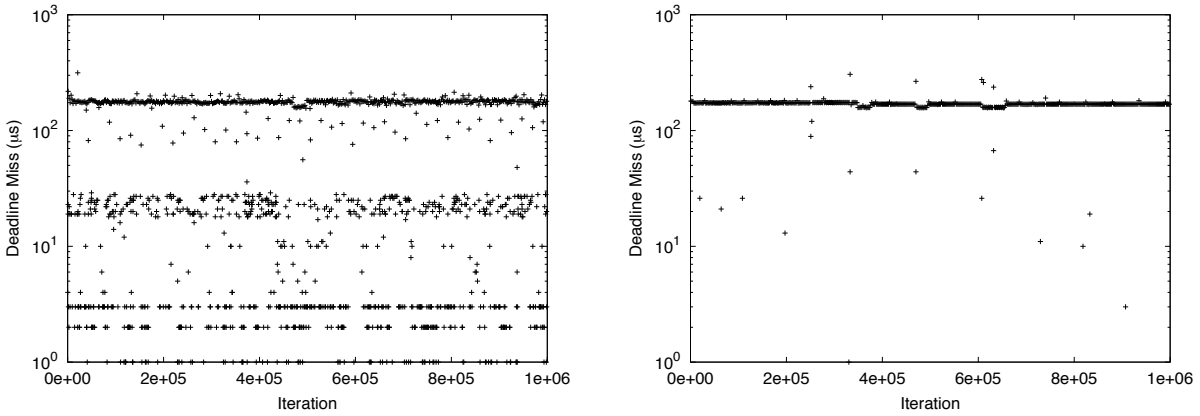


Figure 6.6: Missed deadlines over time for respectively a (a) Reflex and (b) C variant of an audio processing task scheduled for 45 μs periods. The x-axis shows the periodic executions (only 1 million iterations shown) of the time-critical task and the y-axis shows the logarithm of the size of the deadline misses.

the time. Of particular interest in Fig. 6.6 is to see how the perturbation causes regular deadline misses around 180 μs. We consider these anomalies to most likely be caused by buffering on the sound device or to stem from other interactions with the underlying operating system, and we have learned (through private conversations) from the Eventrons project that they experienced equivalent behavior when running at these frequencies. With Reflexes, however, there seems to be further frequent deadline misses in the ranges 2-3 μs, 5-6 μs and around 110-120 μs. These we attribute to the jitter in timeliness as described earlier and depicted in Fig. 6.3 and which also appears to cause similar missed deadlines as seen in Fig. 6.4.

6.4 Evaluation: Stream Processing Graphs

To evaluate Reflex graphs consisting of more than a single task, we considered two benchmark applications developed at MIT for the StreamIt project [TKA02], which we modified to make use of the Reflex API. The benchmark applications used were (1) a beam-form calculation on a set of inputs, and (2) a filter bank for multirate signal processing.[3] Fig. 6.7 shows a graphical representation of the Reflex implementation of the BeamFormer benchmark. It shows the structure and number of tasks as well as their interconnections.

6.4.1 Predictability

To evaluate predictability of a Reflex graph with more than a single task, we considered the `SerializedBeamFormer` benchmark application mentioned above, which we modified by schedul-

[3] A description as well as the actual code for both the utilized StreamIt benchmark applications, `SerializedBeamFormer.str` and `FilterBankNew.str` are available for download at `http://cag.csail.mit.edu/streamit`.

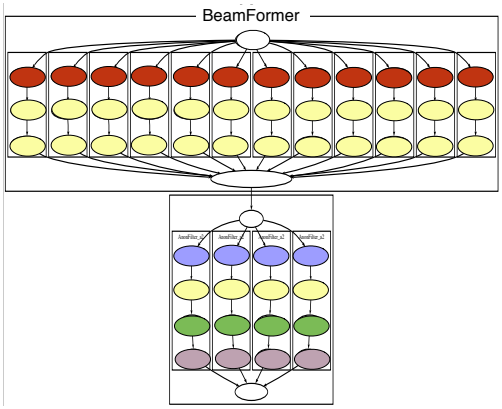
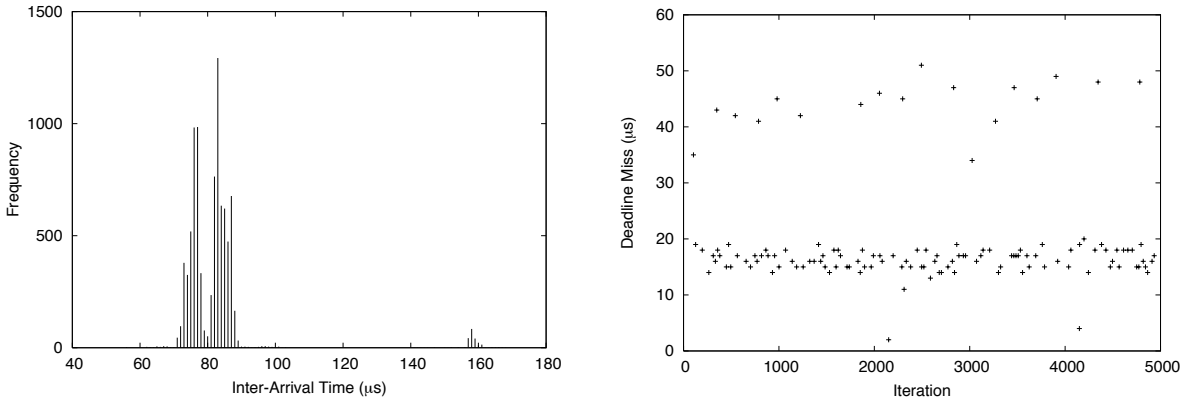


Figure 6.7: Structure of the Reflex graph for the BeamFormer benchmark.



(a) (b)

Figure 6.8: Running Reflex implementation of `SerializedBeamFormer` with periodic thread scheduled every 80 μs over 10,000 iterations. (a) depicts frequencies of inter-arrival time. The x-axes depict the inter-arrival time of two consecutive executions in microseconds of the periodic task whereas the y-axis depicts the frequency, (b) shows missed deadlines over time (5,000 depicted). The x-axis depicts iterations of the task whereas the y-axis shows the deadline misses in μs.

ing the entry task, a void splitter task, with a period of 80 μs instead of being executed continuously. The graph was executed for 10,000 periods on an AMD Athlon 64 X2 Dual Core processor 4400+ with 2GB of physical memory. The operating system used was Linux (kernel version 2.6.17-hrt-dyntick5), extended with high resolution timer (HRT) patches [RT] configured with a tick period of 1 μs.

As depicted in Fig. 6.8(a), nearly all interesting observations of the inter-arrival time are centered around the 80 μs period with only a few microseconds of jitter. This is as it should be considering that the average iteration time of the benchmark is to be around 50 μs, leaving sufficient time for the underlying virtual machine to prepare and schedule the next period. In addition to the

expected peak around 80 μs, there is a number of outliers around 160 μs. We attribute these perturbations to coincidental measurement noise, probably caused by buffering or flushing in the underlying operating system.

Fig. 6.8(b) depicts missed deadlines over time for the Reflex benchmark application. Specifically, out of 10,000 periodic executions, we observed 223 missed deadlines, corresponding to a miss-rate of 2%. The missed deadlines are primarily centered around a range between 15-20 μs throughout the iterations. Most likely, these missed deadlines are a consequence of a slight jitter in the inter-arrival time, as depicted in Fig. 6.8(a). Additionally, Fig. 6.8(b) also conveys a few observations randomly scattered around 30-50 μs. These deadline misses are directly linked with the outlier observations of inter-arrival time around 160 μs in that, generally speaking, a deadline miss between two consecutive periodic executions can cause for the inter-arrival time of the two to be larger than twice the actual period, as explained in Sec. 6.1.

6.4.2 Performance

Both benchmark applications were configured to execute in a uni-processor, single-threaded mode, and thus did not take advantage of the parallelization possibilities of the stream programming paradigm. The performance experiments were performed on a 3.8Ghz Pentium 4, with 4GB of physical memory. The operating system used was Linux (vanilla kernel, version 2.6.15-27-server).

For the sake of comparison, we performed baseline measurements on the automatically generated Java variants of the StreamIt benchmark applications. The Java variants were benchmarked both on the Ovm virtual machine as well as the Java HotSpot virtual machine, version 1.5.0_10-b03, in mixed mode. Reported values are for the third run of the benchmark.

	Reflex/Ovm	StreamIt/Ovm	StreamIt/HotSpot
BeamFormer	314 *ms*	1285 *ms*	1282 *ms*
FilterBank	1260 *ms*	4350 *ms*	3213 *ms*

Table 6.1: Performance measurements showing actual run-time in milliseconds of performing 10,000 iterations of the benchmark applications using respectively Reflex and the Java variants of StreamIt on the Ovm virtual machine and on the Java HotSpot virtual machine.

As depicted in Tab. 6.1, Reflexes perform significantly better than the StreamIt/Java variant also executed on Ovm. Specifically, the performance improvement amounts to a factor 3.5 to 4. It is interesting to compare Ovm and Hotspot. Looking at the results for the StreamIt code, we see that HotSpot is somewhat faster (25%) than Ovm for `FilterBank`. The slowdown can be in part explained by the fact that HotSpot is a more mature infrastructure and also because of known inefficiency in Ovm's treatment of floating point operations, which these applications make heavy use of. It is interesting to observe that Reflex is a factor 2.5-4 times faster than the StreamIt running on HotSpot. This underlines that the significant performance gains achieved are not caused by the virtual machine itself.

6.5 Evaluation: Intrusion Detector System

Finally, we also performed various measurements of our Intrusion Detector System implementation, described in Sec. 3.6, on the Ovm virtual machine again configured with a heap size of 512MB, and running on an AMD Athlon 64 X2 Dual Core processor 4400+ with 2GB of physical memory. The operating system used was Linux (kernel version 2.6.17-hrt-dyntick5), extended with high resolution timer (HRT) patches [RT] configured with a tick period of 1 μs.

The timing of the Reflex graph was configured with a period of 80 μs, meaning that the `PacketReader` creates capsules at a rate of 12.5 KHz. At this rate, the packet synthesizer, running as an ordinary Java thread, is able to generate packets in to the attack detection pipeline without experiencing any underruns. In other words, at a rate of 12.5 KHz the synthesizer can provide packets at the rate which matches the rate with which the Intrusion Detector System can analyze them. The time used to analyze a single network packet (from the capsule creation to the end of the `TearDrop` task) varies from 3 μs to 23 μs with an average of 6 μs. One reason for this variation is that some packets are identified as possible suspects by one of the tasks, and thus require additional processing in the automata. If we consider raw bytes instead of network packets, the intrusion detection system implemented using the Reflex API delivers an analysis rate of 77MB per second.

Part IV

Integration

7 Flexotask Integration

This chapter describes the unified, restricted programming model, called *Flexotask*, resulting from integrating Reflexes, StreamFlex (here considered separately from Reflexes) with Eventrons and Exotasks from IBM Research. The description will mostly be limited to the impact that the integration has had on the Reflex programming model, as described earlier, i.e., the changes we have applied to Reflexes in order to integrate it with the other models. As such, functionality in the Flexotask programming model adopted uniquely from the Eventrons and Exotasks programming models will not be described further as that work is uniquely attributed to their respective inventors. The chapter finishes off by providing code excerpts of a challenging real-time avionics collision detector application demonstrating how to program with Flexotask.

7.1 Motivation

Pursuing the integration of these programming models was motivated by the following challenges. First, the Reflex programming model, like that of Eventrons and Exotasks, represents a static trade-off between expressiveness and latency. By integrating Reflexes with other programming models making other trade-offs, we wanted to have a unified powerful programming model, enabling the programmer to select between these trade-offs within the same programming model, rather than having to choose between different programming models.

Second, we wanted to investigate the viability of porting Reflexes to another virtual machine, without special research related features, such as the preemptible atomic regions, and thereby demonstrate that the Reflex approach is virtual machine agnostic. Moreover, we wanted to explore how Reflexes performed on an industrial-strength virtual machine. As both Eventrons and Exotasks are implemented on the IBM WebSphere Real-Time VM, a virtual machine with

multi-processor design and which comes with its own RTSJ-implementation, we decided to opt for this platform as target, noting the extra challenges and possibilities of the multi-processor support.

7.2 Model Unification

As mentioned, just like Eventrons and Exotasks, Reflexes and StreamFlex represent a static set of trade-offs between expressiveness and latency appropriate to applications with different requirements. Having four different programming models, a programmer would be forced to choose between the different models in order to accommodate the application requirements in the best possible manner. Thus, should the application requirements at some point change, the programmer would be faced with the task of switching API and rewriting the time-critical parts of the application according to the new restricted programming model.

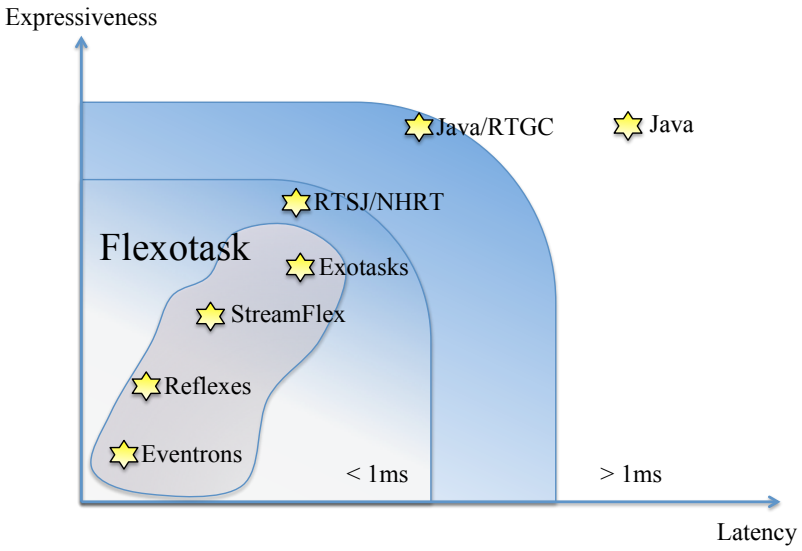


Figure 7.1: The design space of programming models illustrating how the different restricted programming models relate to each other, and the scope of the Flexotask programming model subsuming the four existing programming models.

Consequently, by subsuming these four complementary programming models into a unified programming model, the programmer would get tremendous benefits and flexibility, in that he would have a single model from which to choose various features without being forced to learn a new API and rewrite the application, should the application requirements change. In continuation of Fig. 2.2, Fig. 7.1 illustrates the scope of the Flexotask programming model, covering Reflexes, StreamFlex with Eventrons and Exotasks, enabling the programmer to adjust the level of expressiveness and latency within this area by choosing between various features of the unified programming model.

Feature	Eventrons	Reflexes	Exotasks	StreamFlex	Flexotask
Restricted Unit	Task	Task	Graph	Graph	*Graph*
Long-term Storage	Pre-allocated	Stable region	Private heap	Stable region	*Private heap*
Short-term Storage	Stack variables	Transient region	Private heap	Transient region	*Transient region*
External Comm.	Shared scalars	Transactions	None	Transactions	*Shared and transactions*
Synchronization	Forbidden	Discouraged	Disabled	Discouraged	*Allow, forbid, or disable*
Scheduling	Periodic	Periodic	Pluggable	Data Driven	*Pluggable*
Construction	Direct	Direct	Via template	Direct	*Via template*
Intertask Comm.	–	–	By Deep Copy	By Reference	*Copy or reference*
Enforcement	Initialization	Compilation	Initialization	Compilation	*Compilation + initial.*

Figure 7.2: Features of the four programming models unified into Flexotask.

We used two main strategies to accomplish this unification. First, where a feature of one model was general enough already to subsume another, the more general feature was chosen, and, similarly, less restrictive rules were preferred over more restrictive ones. In choosing the more general or less restrictive capability, we were aware that the less general or more restrictive one may have had advantages of simplicity or efficiency. To recover simplicity for programmers who desire it, we rely on selective veneer interfaces that provide a simplified semantics (for both Eventrons and Reflexes). To recover efficiency for applications that can live within tighter restrictions, we made some stronger checks available but optional (both allocation and synchronization are allowed by default but may be forbidden by static checking).

Second, where two precursor models simply did something differently, we incorporated both mechanisms, and either required a choice (if they conflicted) or allowed both to be used together (if they did not). Thus, we retained the storage management semantics of both Exotasks and Reflexes, the external communication mechanisms of both Eventrons and Reflexes, and the intertask communication mechanisms of both Exotasks and StreamFlex. Fig. 7.2 summarizes some key features of the four models unified as Flexotask, and depicts the feature choices made for Flexotask.

7.3 Introduction to Flexotasks

Being an integration, Flexotask has an interface which is a mixture of the four models. Rather than introducing the entire API, this section will only highlight those parts of the model and API necessary to understand the differences concerning the functionality inherited from Reflexes, including those parts needed to get a consistent picture of Flexotask.

7.3.1 Flexotask Graph

Just like in Reflexes, Flexotask operates with a graph of tasks. Unlike a Reflex graph, a Flexotask graph is constructed indirectly by using a *template* that specifies the layout of the graph. This template can be constructed programmatically or graphically using development tool support (and stored in a special file format), as will be described later.

The template idea, which stems from Exotasks, facilitates the transfer of information about complex programs from development time to run-time, and the independent development of tools that help in the construction of such programs. Fig. 7.3 and Fig. 7.4 shows excerpts of code constructing a `FlexotaskGraph` by programmatically constructing a graph template and through the use of a file containing a preconstructed graph template.

```
FlexotaskTemplate spec = new FlexotaskTemplate();
spec.setStableMode(StableMode.AUTO);
...
Map parameters = new HashMap();
...
FlexotaskGraph graph = spec.validate(SCHEDULER, parameters);
graph.getRunner().start();
```

Figure 7.3: Constructing a Flexotask graph by programmatically creating a graph template.

```
InputStream in = ...
FlexotaskTemplate spec = FlexotaskXMLParser.parseStream(in);
...
Map parameters = new HashMap();
...
FlexotaskGraph graph = spec.validate(SCHEDULER, parameters);
graph.getRunner().start();
```

Figure 7.4: Constructing a Flexotask graph by help of a template preconstructed using the development tool support of Flexotask.

There are some key elements worth pointing out in Fig. 7.3 and Fig. 7.4. The graph template only specifies how the graph should look like – it is not the runnable graph. The actual runnable graph, the `FlexotaskGraph` instance, is returned as a result of the successful validation of the Flexotask graph template against the safety checks. These checks are performed by the Flexotask run-time engine upon invoking the `validate` method on the graph template. This validation occurs at run-time, but before the graph has been started; we use *initialization time* to refer to that point in time. Having passed validation, the tasks in the graph can be started by invoking (indirectly) the `start` method on runner of the `FlexotaskGraph` instance. Details of this initialization time validation will appear later.

7.3.2 Flexotask Task

A Flexotask graph is made up of a number of tasks. The Flexotask programming model distinguishes between a `Flexotask` and an `AtomicFlexotask`, the only difference being that the latter can and must declare one or more methods with transactional semantics that can be invoked by ordinary Java threads. The former can declare no transactional methods, and thus not

share data with ordinary Java threads through this facility. More specifically, the `Flexotask` is an interface that has to be implemented by a class providing task specific logic. Contrary, the `AtomicFlexotask` is an abstract class that implements the `Flexotask` interface and must be subclassed with task specific logic. Fig. 7.5 and Fig. 7.6 show respectively the `Flexotask` interface and the `AtomicFlexotask` abstract class.

```
public interface Flexotask extends Stable {

  public void initialize(FlexotaskInputPort[] inputPorts,
                         FlexotaskOutputPort[] outputPorts, Object parameter);
  public void execute();
}
```

Figure 7.5: The `Flexotask` interface to be implemented by classes, whose instances are to be executed as a time-critical task.

Note how a task in both Reflexes and Flexotask has the same two methods – the `initialize` method used for task to prepare itself before execution starts, and the `execute` method invoked whenever the task is schedulable. The usage of the arguments provided to the `initialize` method are explained later.

```
public abstract class AtomicFlexotask implements Flexotask {

  ...
}
```

Figure 7.6: The abstract `AtomicFlexotask` class to be extended by classes, whose instances are to be executed as a time-critical task. Contrary to class implementing the `Flexotask` interface, any subclass of `AtomicFlexotask` can and must declare transactional methods reachable to the ordinary Java threads. Note, although the abstract class appears to be empty, it is not. Rather it only contains fields and method necessary for the internal functionality, i.e., not methods nor fields that are part of the programming model.

Although the `AtomicFlexotask` class, seen in Fig. 7.6, appears to be empty, this is actually not the case. Rather the class only contains fields and method necessary for the internals of the task, i.e., it does not provide any methods or fields that are part of the programming model. Later, we will describe part of the internal functionality relating to the implementation of the transactional methods.

Like graphs, tasks are represented by templates themselves describing the implementing class of the task, its logical name etc. With this scheme, tasks are also created indirectly along with the rest of the graph during validation using a reflective scheme. Once described through a template, the individual tasks are then added to the graph template, as depicted in Fig. 7.7.

```
FlexotaskTemplate spec = ...
...
FlexotaskTaskTemplate task = new FlexotaskTaskTemplate();
task.setImplementationClass(PacketReader.class.getName());
task.setName("PacketReader");
spec.getTasks().add(task);
```

Figure 7.7: Programmatically adding a Flexotask task to the graph template. Like graphs, tasks are represented by a template that describe the task, e.g., providing the implementation class of the task as well as its logical name.

7.3.3 Memory Management

Similar to a Reflex task, a Flexotask task lives in a fixed size private memory area free of interference from the public heap garbage collection. In Flexotask this private memory area can have two configurations; either be partitioned in a stable and transient area, or only contain a stable area.

By default, the private memory area is not partitioned. Rather, inspired by Exotasks, the entire memory area is configured as a *stable heap*, and in addition has its own garbage collector running independently of the public heap garbage collector. As argued in Sec. 3.3.3, although this approach is attractive from a programming standpoint and efficient compared to a general garbage collection scheme given the limited size of the object graph of the Flexotask, there might be situations where one want to eliminate any avoidable overhead in order to minimize the latencies as much as possible.

For the types of applications requiring the lowest latency requirements possible, Flexotask also features the stable/transient distinction introduced with Reflexes, described in Sec. 3.3.3, in which the transient area is used as an execution scratchpad whose allocations are reclaimed after each execution. In Flexotask, the stable/transient distinction is selected on a per-graph level through the graph template, using either the development tool support or programatically by invoking the `setStableMode` method on the `FlexotaskTemplate` class, as seen in Fig. 7.3. Flexotask operates with several stable modes of which the `DEFAULT` represents the mode where no distinction between stable and transient classes should be observed, and thus the private memory area should only be configured with a stable heap. In other words, in `DEFAULT` mode any allocations made within the Flexotask task are considered stable and destined for the stable heap. In contrast, the mode `AUTO` specifies a Reflex style distinction between stable/transient classes based on the `Stable` marker interface, and of course causing the private memory area to be divided into a stable heap and a transient area.[1] Unlike Reflexes, in Flexotask the programmer does not have to declare the individual sizes of the stable heap and the transient area. Rather,

[1] In fact, Flexotask operates with two other stable modes that really only differ in how stable classes are resolved. Whereas `AUTO` uses the explicit declaration of a stable class using the `Stable` marker interface, the `INFER` mode infers stable classes from the usage of the classes, and `MANUAL` mode expects and follows a manual, programmer-provided list declaring, which classes to be considered stable.

the stable heap and the transient area simply share the memory available within the private memory area, and so the programmer is simply left with the task of estimating the size of the area as a whole. By default, each Flexotask private memory area is given a default size that can be overridden by the programmer.

Even when using the stable/transient distinction, the stable heap is by default garbage collected with its own garbage collector, meaning that although Flexotask adopts the Reflex terminology for stable objects, this does not invariably mean that stable objects are long-lived as in the Reflex model. Rather, the stable heap garbage collector runs on either a scheduled basis, when the task is not running, or on-demand, if memory is exhausted during task execution. By applying a garbage collector to the stable heap (combined with the constant time reclamation of transient objects), the likelihood of ever running out of memory significantly decreases. At the same time, this approach is efficient as the majority of allocations made during an execution are assumed to occur in the transient area, and the root set of the stable heap is thus very limited. Of course, even though this approach is efficient, the programmer still has to take into account the time needed to do a worst-case garbage collection within the task when choosing the period with which the task should run. Since the garbage collection runs in succession of the `execute` method, the period must be chosen such that there is enough idle time to perform a garbage collection without missing deadlines. Full compliance with the Reflex model can, however, be achieved as garbage collection of the stable heap programmatically can be disabled by the programmer if the application has a steady state that does not require reclamation of stable heap data.

7.4 Communication Differences

The Flexotask programming model enables communication between tasks in a graph, and with ordinary threads through transactional methods and static variables, just like with Reflexes. However, there are are minor differences in terms of semantics that are worth highlighting. Furthermore, the Flexotask programming model enables a third way of communicating through *shared instance objects*.

7.4.1 Ports and Channels

In Flexotask, tasks communicate amongst each other using *ports* and *connections*. Each task has zero or more *input ports* and *output ports*. Each port has a data typed, specified with a Java `Class` object. A connection is responsible for binding together an output port on one task with the input port on another, both having the identical type. Although this approach at first sounds similar to that used in Reflexes, there are some key differences between inter-task communication in Reflexes and in Flexotask.

Whereas in Reflexes channels are constructed to hold a limited but flexible number of values, connections in Flexotask are stateless. By default, each output and input port can hold a single value only. However, Flexotask facilitates buffering of the ports such that they can hold multiple

values received or to be sent. In particular this is relevant for data driven applications. Contrary to Reflexes, a buffered port can hold as many values as can be stored within the stable heap of the Flexotask task in which it lives, and is thus not bounded in size.

Like Reflexes, Flexotask facilitates zero copy communication by simply just passing along references to objects over the connections, as illustrated in Fig. 7.8. However, compared to the Reflex programming model, there are key differences. Rather than being restricted to `Capsule` subtypes constrained to only containing fields of primitive and primitive array types, in Flexotask we exploit the fact that we perform initialization time checking, which is more precise than static checking, and thus allow zero copy communication of objects of any reference-immutable between the tasks. These objects to be passed around between tasks no longer live in a special memory area and are no longer maintained by the run-time engine, unlike in Reflexes. Rather, these objects are created externally to the Flexotask graph (on the public heap) and are simple reference-immutable objects that are reachable to the tasks in the Flexotask graph – either through static variables or through so-called *shared instance variables*, as will be described shortly. Like any other heap-allocated reference-immutable object, however, they are required to be pinned.

Besides facilitating zero copy communication, Flexotask also adopts the Exotask style of communication by deep copying values between tasks. This type of communication is, however, restricted to connections of stable types only (as it would be a violation to the static safety checks for a stable type to reference a transient object). Although deep copying is expensive, note, however, that this approach nevertheless effectively avoids any aliasing, which would otherwise compromise the isolation requirements of the tasks.

Like other elements in the Flexotask graph, ports and connections are represented through templates. The run-time representation of these, or rather the array of already connected in- and output ports, are created by the Flexotask run-time engine and passed to the task through its `initialize` method, as seen in Fig. 7.5. The programmer is hereafter responsible for assigning them to appropriate instance fields.

7.4.2 Transactional Methods

A subtle difference between Flexotask and Reflexes is that they use different approaches to declare transactional methods. As mentioned earlier, in Reflexes a transactional method is declared directly on the `ReflexTask` class and is required to be annotated with `@atomic`, an annotation informing the Ovm compiler to execute the method in a preemptible atomic region. Flexotask uses no `@atomic` annotation, but rather requires any transactional method to be declared in a separate interface that extends the `ExternalMethods` interface provided by the Flexotask API, as seen in Fig. 7.9. The reason for introducing this interface is that Flexotask operates with an extra level of isolation by using a delegation pattern [GHJV95], as will be detailed later, and the delegate class will implement this `ExternalMethods` subinterface – exposing to the ordinary Java thread only what is truly reachable.

Following from the requirement to declare all transactional methods in a separate interface,

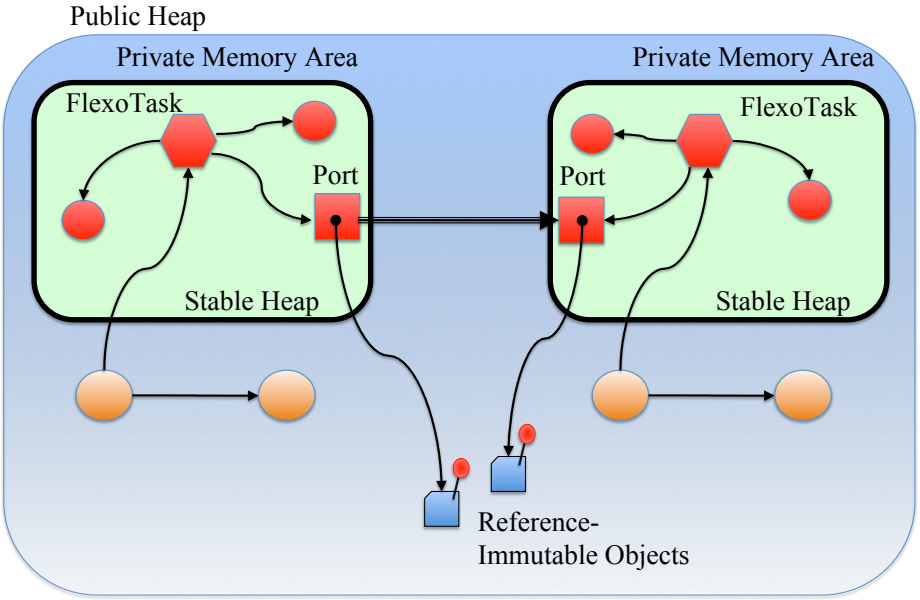


Figure 7.8: Inter-task communication in Flexotask using single-stage buffer ports (red squares), unidirectional connections. Flexotask supports communication with zero copy semantics using heap-allocated, reference-immutable objects that are pinned (depicted) as well as deep copy semantics using stable types only.

```
public interface PacketReaderIntf extends ExternalMethods {

  public void write(byte[] b) throws AtomicException;
}
```

Figure 7.9: Extending the `ExternalMethods` interface to declare methods on a Flexotask that are reachable to and can be invoked by ordinary Java threads with transactional semantics. Figure shows how a Flexotask variant of the `PacketReader` task, seen in Fig. 3.20, would declare its transactional methods.

the class declaration of the Flexotask task must use this interface. Continuing the Flexotask variant of the `PacketReader` example, Fig. 7.10 shows how a Flexotask task would use this interface. Note, since the Flexotask task exposes transactional methods (by implementing the `ExternalMethods` subinterface), the actual Flexotask task class is required to extend the `AtomicFlexotask` class rather than implementing the `Flexotask` interface.

As can also be observed in Fig. 7.9, all methods declared on this `ExternalMethods` subinterface are required to declare that they throw an `AtomicException`. This requirement follows a subtle difference in the transaction semantics between Flexotask and Reflexes. In Reflexes, an abort of a transactional method is transparent to the programmer; the programmer will never know if/when

```
public class PacketReader extends AtomicFlexotask implements PacketReaderIntf {

  ...
  public void write(byte[] b) throws AtomicException {
    buffer.write(b);
  }
  ...
}
```

Figure 7.10: Using the `PacketReaderIntf` interface of Fig. 7.9 in the class declaration of a Flexotask variant of the `PacketReader` task, seen in Fig. 3.20, to declare and implement the transactional methods to be invoked by ordinary Java threads.

the invocation of a transactional method was aborted. Rather, the scheduler will implicitly re-invoke the transactional method until it eventually succeeds. Contrary, in Flexotask the programmer is forced to be aware of the fact that an abort can happen, and must explicitly handle `AtomicException`s indicating that the invocation of the transactional method aborted (as the time-critical task was released before the transactional method committed). Typically, though, this exception handling simply comes down to either retrying the transaction or giving up; an option the programmer does not have with Reflexes, but which might be convenient. However, by making the abort explicit in the programming model, the programmer has more control over the transactional method, whereby the potential unboundedness of the transactional method approach for Reflexes can be avoided.

An evolutionary change from Reflexes to Flexotask stems from the realization that transient objects allocated during the invocation of a transactional method do not really mandate a special allocation area. In fact, from the point of view of the ordinary Java thread, transient objects are fundamentally just like any other heap-allocated object. Consequently, in Flexotask transient objects instantiated by an ordinary Java thread during the invocation of a transactional method are simply allocated on the public heap. Thus, Flexotask still distinguishes between the object lifetime of stable and transient objects, but the allocation context of transient objects is the public heap, as depicted in Fig. 7.11. Note, transient objects allocated during the invocation of the Flexotask task's `execute` method are still allocated in the transient area, if the stable/-transient distinction is observed; the change only affects transient objects allocated during the invocation of a transactional method.

There are several benefits of this change. First, when estimating the size of the private memory area, compared to Reflexes, with Flexotask the programmer has one factor less to worry about. Ignoring the memory requirements needed for the transient allocations made during the invocation of transactional methods, the programmer can concentrate on the memory requirements of the `execute` method. As for the transactional methods themselves, they have the whole public heap at their disposal to allocate transient objects, and any transient object allocations made will become normal garbage (as soon as the invocation of the transactional method exits) that will be reclaimed by the public heap garbage collector. Second, the programmer is not forced to observe

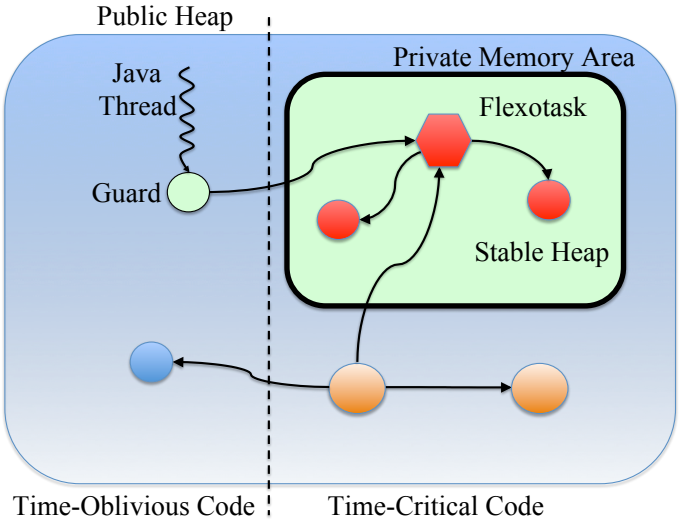


Figure 7.11: Communicating between time-oblivious, ordinary Java threads and a time-critical task through transactional methods. In principle, the communication scheme for Flexotask is equivalent with that of Reflexes, but there are two differences. First, the allocation context of transient objects allocated during a transactional method invocation is no longer the transient area of the task, but rather the public heap (as illustrated). Second, in Flexotask the ordinary Java thread has no direct reference to the time-critical task. Instead, a *guard* (green) is used as a delegate through which ordinary Java threads can invoke the transactional methods that are then delegated to the task running in the private memory area.

a stable/transient distinction within the time-critical task in order prevent a lot of garbage to fill up the stable heap (as without the stable/transient distinction, objects allocated during the transactional method invocation would be allocated as stable objects). Rather, the programmer can configure the task with just a stable heap, yet still allow for transactional methods to be executed without worrying about memory. Finally, by heap allocating transient objects, the necessity to pin any heap-allocated objects used as arguments for transactional methods becomes void. Since transient objects are now also heap-allocated, and thus reachable to the garbage collector, if the garbage collector moves the argument provided to the transactional method, any reference to it from a transient object will be adjusted accordingly.

Note, that changing the allocation context of transient objects (in transactional methods) from a dedicated area to the public heap, does not mean that we abandon the distinction between a transient object (allocated on the heap) and other heap-allocated objects that are part of the time-oblivious code. In fact, for type safety reasons, we maintain this distinction, and furthermore although they are now heap-allocated, transient objects are still subject to the same restrictions imposed by the static safety checks.

An interesting question at this point is, why not change allocation context for all transient objects to the public heap, i.e., not only those allocated in transactional methods but also those

allocated during the invocation of the Flexotask's `execute` method. Unfortunately, there are significant differences that makes this problematic. For instance, the time-critical Flexotask might be executing with such a high frequency and thereby generate so many transient objects (that immediately become garbage) that the public heap garbage collector might find itself unable to keep up. Ultimately, this would lead to an exhaustion of the public heap, forcing a garbage collection of the public heap that will be impossible for the Flexotask to avoid being interfered by. Thus, we maintain the necessity for a dedicated transient area with constant time reclamation for those transient objects instantiated by the real-time thread executing the task.

7.4.3 Guard

As mentioned above, Flexotask uses a delegation pattern for the communication between the time-critical task and ordinary Java threads. This delegate is termed the *guard* and serves one purpose; to effectively ensure isolation of the task by only exposing the transactional methods declared in the `ExternalMethods` subinterface of the `AtomicFlexotask` subclass that it delegates to. Thereby creation of any improper aliases can effectively be prevented.

The guard class is generated automatically by the Flexotask development tools in a process detailed later. The guard is allocated on the public heap and holds a single reference to the time-critical task running in its private memory area such that it can delegate invocations to it. Fig. 7.11 shows the level of indirection introduced through the guard, forcing the ordinary Java thread to invoke transactional methods directly on the guard rather than directly on the task itself, the approach used in Reflexes as illustrated in Fig. 3.12. Consequently, tasks declaring no transactional methods have no guards, and are thus not reachable for ordinary Java threads. Such tasks should simply implement the `Flexotask` interface of the Flexotask API.

Furthermore, as seen in Fig. 7.11, since in Flexotask ordinary Java threads have no direct reference to the time-critical task, an interesting consequence of this shielding is that, unlike a Reflex task, a Flexotask task is not restricted from having non-private instance fields or non-private, non-transactional methods, as detailed later. Although they are public, these fields and methods are not reachable to the ordinary Java thread because of the shielding by the guard.

The guard is instantiated internally by the Flexotask run-time engine during construction of the Flexotask graph, and can be retrieved through the `getGuardObject` on the `FlexotaskGraph` object returned following the successful validation of the graph template. The invocation of the `getGuardObject` must provide the logical name of the task to which to get the guard object for, since multiple tasks in a graph may declare transactional methods, and thus have guard objects. The programmer must cast the guard object to the `ExternalMethods` subinterface implemented by the task, as seen in Fig. 7.12, to invoke the transactional methods on it.

7.4.4 Shared Instance Objects

Besides facilitating communication through static, heap-allocated variables and transactional methods, like is possible in Reflexes, Flexotask also allows for a third way of communicating

```
FlexotaskTemplate spec = ...
...
FlexotaskGraph graph = spec.validate(...);
...
PacketReaderIntf guard = (PacketReaderIntf) graph.getGuardObject("PacketReader");
```

Figure 7.12: Excerpt code showing how to retrieve the guard of the time-critical task with the logical name "PacketReader", a Flexotask variant of the `PacketReader` task seen in Fig. 3.20. Note, how the guard object is referenced through the `ExternalMethods` subinterface, in this case `PacketReaderIntf` – declaration hereof seen in Fig. 7.9.

using *shared instance object*, an approach inherited from Exotasks.

Shared instance objects are heap-allocated objects that can be shared between a Flexotask task and an ordinary Java thread. These shared instance objects are subject to the same restrictions as static variables in that they are required to be (recursively) reference-immutable object graphs, as specified by the definition given in Sec. 4.4. However, shared instance objects are not acquired through static variables.

Rather, in Flexotask references to shared instance objects can also be passed to each task at run-time as part of its initialization, see the `parameter` parameter of the `initialize` method in Fig. 7.5. Parameters were not a feature of any of the precursor models but created explicitly for Flexotask, although Exotasks later have adopted the idea too in a later version [IBM]. Since this passing of the parameter to the individual task cannot happen directly as the task creation is managed by the Flexotask run-time engine, it has to go indirectly through a *parameter map*. The parameter map is a mapping from the logical task name to the shared instance object to be passed to its `parameter`. The parameter map is passed to the `validate` method on the `FlexotaskTemplate`, as seen in Fig. 7.3 and Fig. 7.4, and the Flexotask run-time engine will ensure that each parameter in the map is passed during the initialization to the appropriate task with the correct logical name.

Like any shared heap-allocated objects referenced by a Flexotask task (or for that matter a Reflex task), shared instance objects too are required to be pinned by the virtual machine to their location on the public heap during the time when they are accessible from within the task. In practical terms, this means that the shared instance objects must be pinned throughout the lifetime of the Flexotask graph since they are passed in as part of the `initialize` method to the task, seen in Fig. 7.5.

7.5 Pluggable Scheduling

Scheduling is handled differently in the Flexotask programming model. Rather than being limited to a particular timing semantics like in Reflexes, Flexotask takes a more general approach by promoting a looser coupling between model and timing semantics, an approach adopted from

Exotasks. Central to this decoupling are timing grammars. A timing grammar provides a set of rules for attaching timing annotations to the different elements in the Flexotask graph, e.g., the graph itself, tasks, and the connections. The timing annotations are read and interpreted by a pluggable scheduler that expects annotations on the graph elements conforming to the grammar.

At development time, as part of the graph template, Flexotask facilitates the specification of the timing grammar to be used by a given Flexotask graph. More specifically, timing annotations can be added to the graph itself as well as the elements of the graph. Like with the rest of the graph template, the timing annotations can be specified either programmatically or as part of the graphical editing provided by the development tool support integrated into the Eclipse IDE.

```
FlexotaskTemplate spec = ...
spec.setTimingData(new Period(periodInNanos * 1000));
...
FlexotaskTaskTemplate task = ...
task.setTimingData(new SimpleTimingAnnotation(new long[]{0}));
...
FlexotaskGraph graph = spec.validate("TTScheduler", parameters);
```

Figure 7.13: Excerpt code showing how to programmatically specify the timing annotations on both a Flexotask graph and task – in the example a timing grammar supporting periodic execution is selected. Note, the name "TTScheduler" represents the logical name of the time triggered scheduler.

The timing grammar is supported by an associated scheduler capable of interpreting and realizing the timing requirements of the grammar. At run-time, the timing grammar specified in the template is used to select and load the appropriate scheduler that will be responsible for executing the Flexotask graph.

7.6 Integration of Static Safety Checking

Flexotask operates with two levels of isolation – *strong* and *weak* that can be selected by the programmer. The *strong* isolation model is inherited from Exotasks, and while a part of the Flexotask model, we will not focus more on it here, but only note that this isolation model puts further constraints on the Flexotask task that the static safety checks, described in Sec. 4, by, e.g., prohibiting communication with ordinary Java threads, only allowing communication between tasks by deep copy, and only allowing access to static variables if these are immutable (as opposed to just reference-immutable). Rather, this isolation model is appropriate for applications that are very self-contained and for which a high degree of determinism is required.

The *weak* isolation model in Flexotask is to a large extent equivalent to that of Reflexes. However, there are a few noteworthy variations that have an impact on the static safety checks. Fig. 7.14 illustrates the legal and illegal object references in Flexotask between the time-critical task and ordinary Java code.

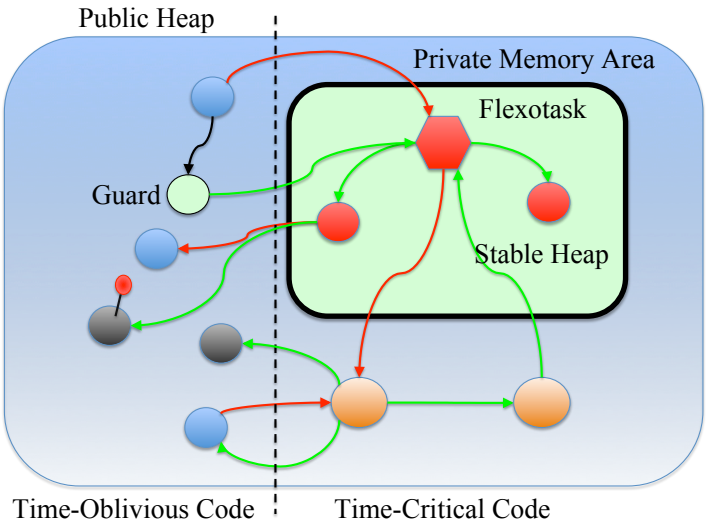


Figure 7.14: The legal and illegal object references in and out of a Flexotask instance that the static safety checks must ensure are respectively allowed and caught. The figure illustrates a Flexotask instance in a private memory area with its object graphs of stable and transient objects as well as a number of heap-allocated objects, including the guard object, of which one is pinned. Object references are illustrated with green and red arrows, representing legal and illegal references respectively. Note that only transient objects allocated during the invocation of a transactional method are heap-allocated. Transient objects allocated by the real-time thread invoking the `execute` method of the task are allocated in the transient area (if the stable/transient distinction is observed).

7.6.1 Implicit Ownership Relaxation

Comparing Fig. 7.14 with Fig. 4.1, it is obvious that the differences in legal/illegal references between Reflexes and Flexotask relate to (1) the introduction of the *guard*, and (2) the decision to change the allocation context for transient objects allocated during the invocation of a transactional method from a dedicated transient area to the heap. The consequence of this latter change does, however, not affect the static safety checks as such, since transient objects have been permitted to reference certain heap-allocated objects all the time. Rather, the change affects the run-time requirement that these heap-allocated objects passed in as arguments to the transactional method have to be pinned to be safely used. This requirement is no longer necessary, as discussed earlier.

Concerning the guard, whereas in Reflexes, the Java application has a direct reference to the `ReflexTask` object, in Flexotask the `AtomicFlexotask` instance is referenced indirectly through its guard (A `Flexotask` instance is not reachable at all since it does not expose any transactional methods). A direct reference to the `AtomicFlexotask` instance is simply not possible to achieve due to the reflective way the Flexotask graph is instantiated by the Flexotask run-time

engine. Rather, the Java application can only reference the guard instance, that in turn then references the `AtomicFlexotask` instance, and, as mentioned earlier, the guard exposes only the transactional methods of the `AtomicFlexotask` declared in its `ExternalMethods` interface.

Realizing the effective shielding provided by the guard, we can take advantage hereof and loosen some of the static safety checks concerning *Implicit Ownership*. More specifically, the static checks $\mathcal{R}3$ and $\mathcal{R}4$, described in Sec. 4.3, become irrelevant. Consequently, in Flexotask, an `AtomicFlexotask` class declaring public fields of reference types or public, non-transactional methods poses no type-safety concerns as these fields and methods are not reachable from the Java application nor from any other tasks in the graph. Of course, since a `Flexotask` is not reachable at all, the same relaxation of the restrictions applies to it.

7.6.2 Capsule Type Relaxation

As mentioned, Flexotask does not operate with a specific `Capsule` type like Reflexes. Rather, we achieve the same functionality by supporting pass-by-reference for reference-immutable objects allocated on the public heap that the Flexotask are allowed to access already – through either static fields or shared instance objects. Thus, with this in mind we can relax the safety checks concerning capsules, in Sec. 4.5. Given the fact that Flexotask performs initialization time checking, and the code thus is subject to a more precise analysis than at development time, we can relax the restriction from Reflexes that capsules can only contain fields of primitive and primitive array types. More precisely, in Flexotask we allow any reference-immutable object to be used for zero copy communication.

$\mathcal{R}12$: *Any heap-allocated instance of a reference-immutable type can be used for zero copy communication.* □

With this change, in Flexotask the run-time engine does not maintain a communication area containing some pool of reusable capsules, like in Reflexes. Thus, in Flexotask we do not have to worry about reclamation of the objects used for communication between tasks in order to reuse them. Rather, the complete set of reference-immutable objects used for communication are passed to one or more tasks during initialization, and remains fixed throughout the lifetime of the Flexotask graph as they are pinned, and therefore will not be reclaimed by the public heap garbage collector. Thus, unlike in Reflexes, there is no danger of having a 'leak' of communication objects, if a task should keep a reference to such an object, as the set of objects is fixed. With this property in mind, we can further relax the safety checks by eliminating $\mathcal{R}13$ completely. In other words, we no longer mandate that objects used for communication are transient, and consequently allow a heap-allocated reference-immutable object to be of stable type and thereby be assignable to a stable field within a task.

7.7 Example: Avionics Collision Detection

To illustrate the usage of Flexotask, we present the example of an avionics collision detection algorithm (based on the source code of [ZNV04]). Collision detection is performed by a single atomic Flexotask, which periodically processes the latest frame it has received. Each frame contains aircraft call signs paired with the positions and direction vectors of the aircraft. The output of the algorithm is a warning each time any pair of aircrafts are on a collision course. In our implementation, the aircraft call signs, positions and direction vectors are all provided by a separately running ordinary Java thread that simulates this data based on symbolic execution of a set of equations describing aircraft trajectories.

```
// Load template from XML−file
InputStream in = DetectorTask.class.getResourceAsStream("Detector.ftg");
FlexotaskTemplate spec = FlexotaskXMLParser.parseStream(in);

// Validate graph and provide sharedArray in parameter map
Map parameters = new HashMap();
RawFrameArray sharedArray = new RawFrameArray();
parameters.put("DetectorTask", sharedArray);
FlexotaskGraph graph = spec.validate("TTScheduler", parameters);

// Get reference to Detector guard
DetectorGuard detector = (DetectorGuard) graph.getGuardObject("DetectorTask");

// Start the graph (i.e., the tasks in the graph)
graph.getRunner().start();

...

// Communicate a new frame from ordinary Java thread to Flexotask
int frameIndex = detector.getFirstFree();
if (frameIndex == −1) { ... no buffer available }
frames.get(frameIndex).copy(lengths, callsigns, positions);
detector.setNextToProcess(frameIndex);
```

Figure 7.15: Constructing a Flexotask graph through an XML-based template constructed with the Flexotask editor integrated into Eclipse.

This example illustrates the simultaneous use of both kinds of external communication, as was discussed in Sec. 3.4.9. Because each frame potentially contains a fair amount of data, we do not incur the overhead of transmitting this information atomically. Rather, we rely on a ring buffer of `RawFrame` data structures, which is shared between the Flexotask and the simulator. These handle the bulk data transfer. The coordination around availability of frames for use by either the Flexotask or the simulator is handled by transactional methods on the Flexotask task. The invariant maintained by these methods is that no frame is ever used simultaneously by both.

The example also illustrates the tradeoff between the use of pure stable heap allocation and stable/transient allocation discussed in Sec. 7.3.3. We were able to get the example working quickly using pure stable heap allocation (in which no classes are declared stable), which was necessary because the `StateTable` needed by the Flexotask used Java collection classes (that of course are not declared stable). Then, to optimize the example, we replaced the Java collection classes with a small number of custom classes that were stable according the rules of Sec. 4.3. In all, this example required six classes to be labelled stable (in addition to the Flexotask itself, which is implicitly stable, and arrays of stable classes, which do not have to be labelled).

A sketch of the main program is shown Fig. 7.15 which illustrates the reconstruction of a template from one previously prepared in the development environment, and stored as an XML file. The `FlexotaskXMLParser` class provides capabilities for loading templates and creating a Java object representation. The template encapsulates the name of the single task ("DetectorTask"), its implementation class (`DetectorTask`), and the list of stable and transient classes that were marked and checked at development time. The `validate()` method selects a scheduler (pluggable, as in the Exotasks [ABI+07] system) and produces a runnable `FlexotaskGraph`. The `parameters` argument passes in the shared instance objects that will be used for communication. The `getGuardObject()` method retrieves a reference to the guard for the task that can be used to invoke task atomic methods by the ordinary Java thread. This object is automatically generated from the `DetectorTask` class and its transactional interface `DetectorGuard`, a process detailed later.

Fig. 7.16 shows the class `DetectorTask` which extends `AtomicFlexotask` and implements `DetectorGuard`. The `initialize` method establishes the sharing relationship by storing the `RawFrameArray` parameter and initializes the `StateTable` to store stable state. In a graph with more than one Flexotask task this method would also pass in representations of the task's ports to use for inter-task communication, and these would be saved in instance variables. The `execute` method establishes the frame to be processed and analyzes the data it contains. Each invocation of `execute` will create transient objects of types `Detector` and its numerous dependent working objects, as well as new stable objects to represent call signs and vectors that will be stored in the `StateTable`. The implementation of the `getFirstFree` and `setNextToProcess)` methods represents the code as the programmer would write it. The actual code executed at runtime is instrumented at the bytecode level so as to redirect all mutations (and subsequent reads thereof) to a transaction log that is then committed by rolling forward the mutations in an epilog, as described later. As such, the transactionality of a method is completely transparent to the programmer, except that the invoking program should catch and handle the `AtomicException`.

As previously mentioned, the `RawFrameArray` data structure must be reference-immutable in order to be safely shared between the ordinary Java thread and the Flexotask task. The code of this class is shown in Fig. 7.17. For reference-immutability to hold, the `RawFrame` data structure must first be reference-immutable, which is easily accomplished since this class is just a set of primitive arrays connected to their parent object by `final` references. But, an array of references is not normally reference-immutable. This problem is solved in the Flexotask system as it is in Eventrons by using a special `ImmutableArray` class. This class's constructor copies its argument and does not subsequently leak it, ensuring that no mutations occur to the enclosed array after

```
public interface DetectorGuard extends ExternalMethods {
  public void setNextToProcess(int frameIndex) throws AtomicException;
  public int getFirstFree() throws AtomicException;
}

class DetectorTask extends AtomicFlexotask implements DetectorGuard {
  private StateTable state;
  private RawFrameArray frames;
  private int nextToProcess;
  private int firstFree;

  public void initialize(..., Object parameter) {
      frames = (RawFrameArray) parameter;
      state = new StateTable();
  }

  public void execute() {
    if (nextToProcess != firstFree) {
      cd = new Detector(state, Constants.GOOD_VOXEL_SIZE);
      cd.setFrame(frames.get(nextToProcess));
      cd.run();
      nextToProcess = firstFree = increment(nextToProcess);
      // increment 'increments' modulo a ring size
    }
  }

  public int getFirstFree() throws AtomicException {
    int check = increment(firstFree);
    if (check == nextToProcess)
      return -1;
    int ans = firstFree;
    firstFree = check;
    return ans;
  }

  public void setNextToProcess(int nextToProcess) throws AtomicException {
    this.nextToProcess = nextToProcess;
  }
}
```

Figure 7.16: The `DetectorTask`, an `AtomicFlexotask` responsible for detecting aircraft collisions.

```
class RawFrameArray implements Stable {
  private final ImmutableArray frames;

  public RawFrame get(final int i) {
    return (RawFrame) frames.get(i);
  }

  public RawFrameArray() {
    RawFrame[] innerArray = new RawFrame[MAX_FRAMES];
    for (int i = 0; i < MAX_FRAMES; i++)
      innerArray[i] = new RawFrame();
    frames = new ImmutableArray(innerArray);
  }
}
```

Figure 7.17: A reference-immutable data structure shared between the ordinary Java thread and the Flexotask task as a shared instance object brought into the task through the `initialize` method.

construction.

8

Flexotask Implementation

This chapter highlights the most interesting aspects of the implementation of Flexotask on top of the IBM WebSphere Real-Time VM. The Flexotask system comes with development tool support integrated in the Eclipse IDE as well as virtual machine support implemented in the IBM WebSphere Real-Time VM.

8.1 Eclipse Integration

The development tool support integrated in the Eclipse IDE provides powerful support for easing the development of the Flexotask applications and enforcing the restrictions imposed by the programming model. In addition, since the IBM WebSphere Real-Time VM does not have support for preemptible atomic regions, like Ovm, the development tool support also facilitates rewriting of bytecodes to enable transactional methods. More specifically, this development tool support is provided through respectively a graphical editor to help the construction of the Flexotask graph templates, and a project builder that is run subsequent to the standard Java compiler to enforce the restrictions. Fig. 8.1 gives an illustration of the overview of the Flexotask infrastructure.

At development time, Flexotask programs are written like any other Java program in the Eclipse IDE. However, to enable full development support, the specific project must be configured as a Flexotask project in Eclipse, a project setting that extends the standard Eclipse Java project.

Part of the development support integrated into Eclipse involves a graphical editor, as depicted in Fig. 8.2, that we adopted from Exotasks and extended. The graphical editor enables the programmer to graphically construct the Flexotask graph, rather than doing all of this pro-

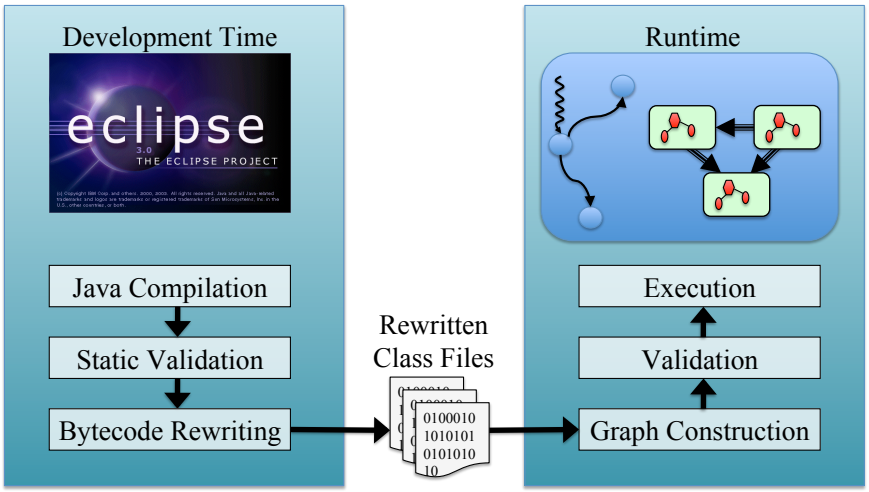


Figure 8.1: Flexotask programs developed within the Eclipse IDE are validated at development time against the type rules. Following successful validation, class files are rewritten to include support for transactions. At initialization time, the Flexotask run-time engine constructs the Flexotask graph and performs a data-sensitive analysis to ensure correctness, after which the graph is executed.

gramatically. Whereas the programmer does not really benefit from this editor support when constructing simple-task programs, more complex graphs involving several tasks and many connections clearly can leverage from this support. The output of this graph construction is an XML-based specification file, that can subsequently be loaded and parsed by the Flexotask run-time engine, as for instance seen in Fig. 7.15.

Following successful standard Java compilation of the Flexotask project, the Flexotask project builder is invoked by Eclipse, causing for the code to be statically validated against the safety checks, described in Sec. 4 and Sec. 7.6. Specifically, the validation is performed on the Java bytecodes of the classes related to the time-critical code. If the code fails to validate correctly, the programmer is notified with a list of errors and warnings in the default *Problems View* of Eclipse. Contrary, if the code verifies correctly, classes reachable through the transactional methods invoked by the ordinary Java threads are bytecode rewritten by the Flexotask builder, a process detailed later.

Together with the classes of the remainder of the Flexotask application, the rewritten classes are then transferred manually by the programmer from the development platform to the run-time platform – in our implementation the IBM WebSphere Real-Time VM.

Then, after the code is loaded into the virtual machine and the graph has been instantiated, the code (including the rewritten classes) is validated again against the safety checks, but this time using run-time information about arguments and static variables. This architecture adopts the best of the precursor models. The Reflex and StreamFlex models performed validation statically at development time, which has the advantage of early detection of errors and convenience for

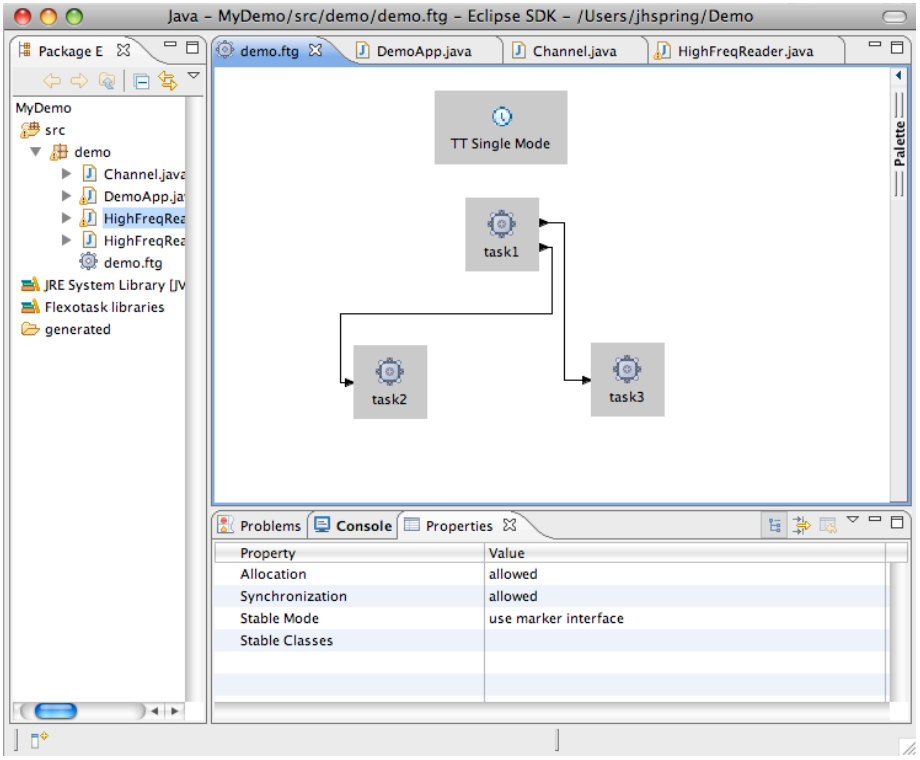


Figure 8.2: The Flexotask graphical editor integrated into the Eclipse IDE. Screenshot shows the graphical creation of a Flexotask graph having three tasks. Note, the palette to the right allowing the selection of components to be dropped in the editing area. In the bottom, the properties of each component can be edited.

the programmer by highlighting exactly the violating lines in the source code. The Eventrons and Exotasks models performed validation at initialization time, i.e., after class initialization and object construction but before run-time, using a data-sensitive analysis that is more precise and admits a larger set of valid programs than the static validation. In addition, static enforcement alone does not prevent untrusted code to be run in a restricted thread since there is no guarantee that the code attempted executed actually went through and passed the appropriate validator. To achieve the advantages of both kinds of checking, in Flexotask we do the checking twice, with some of the (necessarily) more conservative checks during development reduced to the status of warnings, whereas during initialization time checking these same checks result in status of errors. In contrast to our approach where all validation occurs at two stages prior to the actual run-time, the `NoHeapRealtimeThread` employs continuous run-time checking since with `NoHeapRealtimeThread` it is much harder to determine statically whether a program is correct.

8.2 Scheduling

Whereas the Ovm implementation of Reflexes used the scheduler of the virtual machine for scheduling, in Flexotask, we adopted the *pluggable scheduler* approach of the Exotasks model as this represents the most general solution, and thereby giving the scheduler the responsibility to schedule not only tasks, but also data movement between tasks and the garbage collection of tasks. Included in the responsibilities of the scheduler is to perform any garbage collection of the stable heap of the Flexotask tasks. Recall, that the garbage collection of the stable heaps of Flexotask tasks is done on a scheduled basis, or on-demand, in case of a memory shortage.

8.2.1 Time-Triggered Scheduler

We carry forward a time triggered scheduler as from Reflexes, which supports periodic execution of single tasks. The scheduler implementation used in Flexotask is inherited directly from Exotasks, which provides richer support for graphs of tasks, with tasks being assigned specific time offsets within the period. As in Exotasks, all restricted threads in Flexotask actually belong to schedulers and the mapping of threads to tasks is usually not one-to-one (more typically, the number of threads reflects the level of real concurrency available in the hardware).

8.2.2 Scheduler for Stream-Based Applications

In addition to a periodic scheduler, Flexotask also provides support for a data driven scheduler for stream-based applications. Taking advantage of the fact that the IBM WebSphere Real-Time VM supports multi-processors, this scheduler enables the execution of the tasks in the Flexotask graph to be truly parallelized and optimized according to the number of processors available; features not exploited in the time triggered scheduler. In fact, the stream-based scheduler extends the time triggered scheduler in that it supports periodic executions. However, where the time triggered scheduler only uses a single thread to execute single tasks, or simple graphs of tasks, the stream-based scheduler will deploy several threads to execute complex graphs.

Central to this scheduler support for stream-based applications is a timing grammar for stream-based Flexotask programs – `StreamBasedTimingAnnotation`. This timing grammar provides support for annotating the input/output rates of a Flexotask task. More specifically, the rate declarations allow each task to declare the number of buffer elements it consumes, the number of buffer elements to which it peeks, and the number of elements it produces. A rate can be declared as either a single digit integer number or a range of the form *[N,M]* – where $M > N$. In the current design Flexotask does not support parameterized rates.

For parallelizing streams of data, Reflexes provide special tasks for multiplexing/demultiplexing the elements between tasks – the splitter and joiner, as described in Sec. 3.4.4. In Flexotask there are no such special tasks. Instead, the grammar for stream-based applications allows for ports having multiple outgoing or incoming connections attached to specify respectively their mode for splitting and joining the messages. The grammar enables an outgoing port to specify

a policy based on either *round-robin* or *duplicating*, and for an incoming port, *round-robin* only. In other words, fully compatible with splitters and joiners in Reflexes.

Finally, in order for elements to fill up in the various buffers while waiting to be multiplexed/demultiplexed, the timing grammar enables ports to be annotated in order to support buffering. For instance, given a round-robin policy in which two elements are taken from each incoming connection, the single-slot port is not sufficient. A buffered port is unbounded within the size of the task's stable heap.

The `StreamScheduler` is implemented as a pluggable schedulers in the Flexotask framework and is capable of interpreting the `StreamBasedTimingAnnotation`. Devising an optimized schedule for a complex graph is not trivial. Rather than inventing our own algorithms for calculating how many times each task should run, in which order, and on which processor, we exploit the basic optimization algorithms from the StreamIt project [TKA02]. Thus, the `StreamScheduler` uses the StreamIt API to device an optimized schedule. For this to be viable, the Flexotask graph has to go through a number of steps as illustrated in Fig. 8.3.

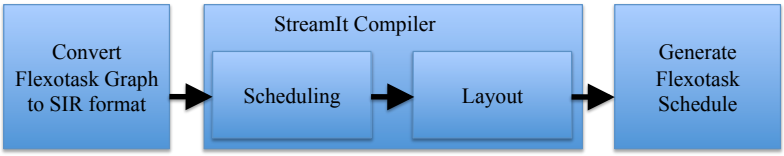


Figure 8.3: The phases of the Flexotask graph in order to create a schedule using the `StreamScheduler`.

The `StreamScheduler` will first convert the Flexotask graph into an equivalent `SIR` graph, the internal representation used by StreamIt to represent tasks. This step is necessary in order to exploit the StreamIt API. Through this conversion, tasks in a Flexotask graph are represented as `SIRFilter` in an equivalent StreamIt graph, and are given the same I/O rates as annotated to the Flexotask tasks. One particular aspect about this conversion is how a Flexotask task with multiple output connections is converted into two `SIRFilter`s; one representing the task logic itself, and one representing the splitter, `SIRSplitter`. Likewise, a Flexotask task having multiple incoming connections will be converted into a joiner, `SIRJoiner`, and a subsequent `SIRFilter` representing the logic.

Once this conversion has taken place, the StreamIt compiler is invoked on the SIR graph in order to perform the calculation of the schedule. For this calculation, the number of processors for the target platform is provided from the Flexotask graph template. In particular, the calculations include the following phases (described in detail in [GTK+02]):[1]

- **Scheduling** – This scheduling phase consists of two parts. First, it performs rate matching

[1] In fact, the StreamIt compiler would perform two additional phases; graph expansion and partitioning. Since we in Flexotask do not support parameterized rates, the graph is, however, already fully expanded. Partitioning involves fission and fusion transformations for load balancing, but since we want to retain a one-to-one mapping from a `SIRFilter` to a Flexotask task, this step is not needed either.

of the `SIRFilter`s, calculating the filter multiplicity scale that tells the scheduler how many times to run each filter. This is useful when running on a processor with a cache-based architecture as cache locality can be exploited. Second, it performs a partitioning of the stream graph into phases rather than a single stream, which from a cache perspective, again, is beneficial as the individual nodes might be executed differently by the scheduler.

- **Layout** – The goal of the layout phase is to assign nodes in the stream graph to computational units on the target platform and orchestrate communication between these units.

Following these optimizations, an optimized schedule is returned. This schedule describes for a single run-through of the tasks in the graph, how many times each task should be run, and on which processors. However, the returned schedule comes in an internal StreamIt representation and must be transformed into a Flexotask specific schedule. Specifically, we map each requested computational unit to a real-time thread in the virtual machine with the knowledge that these will likely be mapped to operating system threads on different processors.

The scheduler observes two run-time phases; a *prime-pump* phase in which each task in the graph is executed in topological order the specific number of time to fill up the buffers, and a *steady-state* phase, which from this point on is used to execute the tasks in the graph. Note, in the steady-state phase the ordering with which tasks in the graph are executed is irrelevant. However, for each graph run, each task in the graph must be executed the number of times in the schedule. This independence between tasks makes it ideal for execution in parallel; the only fix-point between the threads executing the various parts of the graph is the *graph run-through*.

8.3 Unification of Safety Checking

As seen from Fig. 7.2, prior to integrating the four restricted programming models, Reflexes and StreamFlex enforce type safety statically at development time, whereas Eventrons and Exotasks did so at initialization time. Realizing that one approach does not subsume the other, and that both approaches have their advantages, in Flexotask we have implemented a common framework for checking type safety both at development- and initialization time. This framework is embodied in the Flexotask project builder integrated into Eclipse.

This framework is centered around an analysis engine that performs *Rapid Type Analysis* [BS96] to build a summarized call graph rooted in the initially reachable methods of the individual tasks in the Flexotask graph. More specifically, this graph takes starting point in the `execute`, `initialize` methods of all the tasks, in addition to the set of all transactional methods declared in the `ExternalMethods` subinterfaces, if the task is a subclass of `AtomicFlexotask`. From this initial starting point, the analysis engine examines every bytecode of every reachable method using the Apache Byte Code Engineering Library (BCEL) [Apa]. At initialization time, the bytecodes to be analyzed are found in the already loaded and verified classes of the virtual machine (and classloading is forced by the checker to ensure initialization hereof). Contrary, at development time, bytecodes are read directly from the class files, which are loaded from the project output path using a conventional class file parser.

The rules to be enforced include (1) segregation of classes into stable and transient, (2) checking that parameters passed by reference to the tasks are reference-immutable, and (3) checking that references acquired through `static` variables are reference-immutable too. The majority of these checks being performed at development- and initialization time are common for the two. However, one particular analysis which differs relates to inferring reference-immutability.

8.3.1 Initialization Time Checking

The initialization time checker performs its checking for reference-immutability in a *data sensitive* fashion. That is, it maintains the set F of field signatures (static or instance) found to be referenced (for reading or writing) in any method in the Flexotask call graph, and a set O of objects residing on the public heap but accessible to Flexotask code. Initially, the set O consists of objects passed as parameters to the `initialize` methods of the `Flexotask` s, but it is augmented when a `static` field is accessed with a `getstatic` bytecode. Whenever a field signature is added to F, the checker considers objects in O that contain a matching field. The referents of such fields are added to O. Whenever an object is added to O, it is inspected for fields that match F. Thus, an addition to either set can expand both sets up to a fix-point. When objects are added to the O set, they are checked for effective finality. If this criteria is not met, the initialization time checker will cause for an error message describing the safety violation to be printed on the command-line.

Following a successful checking of the Flexotask graph, the initialization time checker has an additional responsibility. The checker must pass a list of stable types to the Flexotask run-time engine, such that these during run-time can be allocated in the stable heaps of the tasks. How the actual list of stable types is constructed depends on the stable mode chosen in the graph template. In the case of `MANUAL` stable mode, the list of stable types comes directly from the graph template itself, where the programmer has declared them. In the three other stable modes, the list is automatically generated by the initialization time checker using either the `Stable` marker interface (`AUTO`), usage inference (`INFER`) or any reached class (`DEFAULT`).

The approach used here to directly inject the list of stable types from the type checker to the Flexotask run-time engine is much safer compared to the approach taken in the prototype, as described in Sec. 5.6, as the programmer cannot manipulate the list between the checking and run-time stages. Recall, in Reflexes, the Reflex run-time engine blindly trusts the list of stable types provided to it through a text file. In Flexotask, the programmer could maliciously try to circumvent safety by manually providing an incorrect list of stable types. However, if unsafe, the initialization time checker would simply reject the program, and the program would never reach a point of execution.

8.3.2 Development Time Checking

The development time checker has two challenges not faced at initialization time. First, it does not know the actual objects passed as parameters (neither to the `initialize` method, nor to any transactional methods), nor does it know the actual objects present in any `static` variables

used by the Flexotask tasks. Instead, the development time checker relies on a class-based analysis that is inevitably more conservative than the data sensitive.

The safety checking concerning the stable/transient distinction does not change as the rules are based on classes rather than objects. However, because the set of potentially live classes inferred by the analysis engine may be larger at development time, some potentially transient classes may not be identified.

Checks for reference-immutability, however, are based on a class-based definition. To find the relevant live classes, the class initialization methods must be examined as well as other methods analyzed by the analysis engine, according to the algorithm seen in Fig. 4.2. If a class is checked for reference-immutability and it can be statically determined not to be reference-immutable, the development time checker will display an error message to the programmer in the *Problems View* of Eclipse. If, however, its live class set cannot be statically bounded and thus it cannot be determined whether or not the class is reference-immutable, the development time checker will instead display a warning message to the programmer. The warning message simply serves to inform the programmer that given the imprecision of the development time checker, some type was found to be suspicious, and it might escalate to become an error during initialization time checking.

A second challenge faced by the development time checker is that of incomplete information. The development time checker analyzes each Flexotask graph template that it finds in the Eclipse project classpath and any classes referenced by it, although this set of classes may be incomplete. It also analyzes (individually) any orphan Flexotask classes not referenced by any template, on the grounds that they may later be so-referenced. It continuously posts error and warning indicators in the *Problems View*, which may be temporarily (or permanently) suppressed by annotations. This is necessary since the initialization time checker, with more information, will be more precise and may permit things that the development time checker flags as suspicious. Fig. 8.4 shows a screenshot of how the Flexotask development time checker reports violations of the static safety checks in its *Problems View*.

Following the successful checking of the code, the development time checker is responsible for applying a set of program transformations on the bytecodes of an `AtomicFlexotask` in order to facilitate for transactional semantics of the methods being invoked by ordinary Java threads. These program transformations are described shortly.

8.4 Pinning of Objects

In Flexotask, identifying which heap-allocated objects to be pinned and performing the actual pinning is the responsibility of the initialization time checker. Whereas our prototype implementation of Reflexes leveraged the standard RTSJ allocation policy of static variables and consequently allocated them in a special memory area never garbage collected, in the Flexotask implementation we do not rely on such an area. Rather, the objects to be pinned are exactly those that the initialization time checker identified in the set O, described earlier. Not included in O are objects passed as arguments to transactional methods.

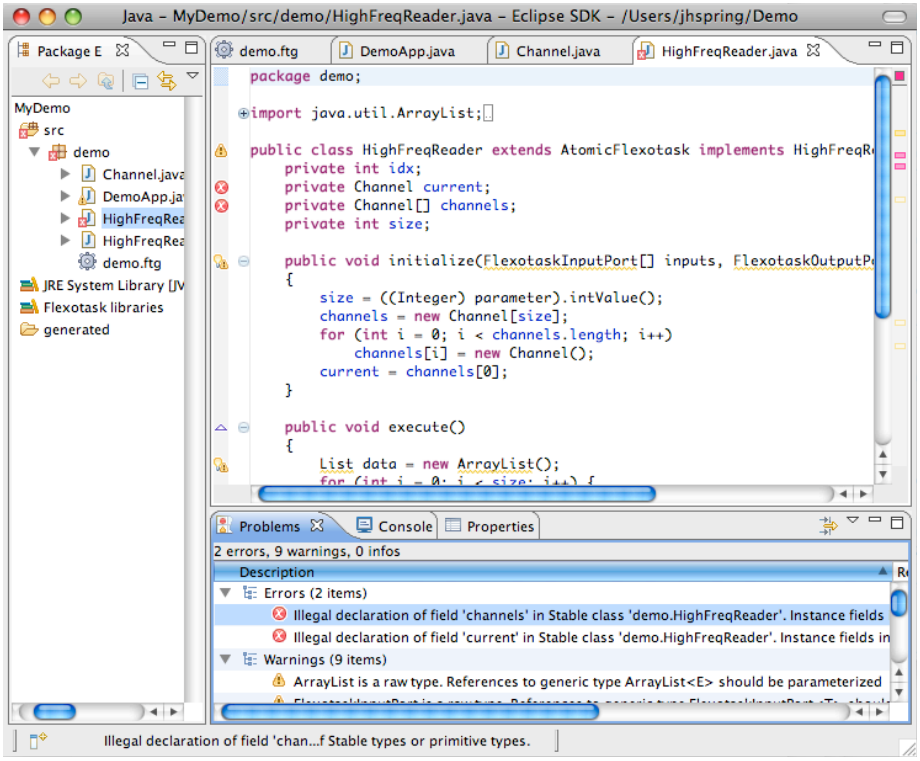


Figure 8.4: The Flexotask development time checker in action. Screenshot shows the development time checker has detected two violations of the static safety checks in the checked code. The errors are reported in the *Problems View* of Eclipse. Note also how the two violating code statements are highlighted with a red marker, making them easy for the programmer to identify. The reported violations concern the fact that the `Channel` class is unexpectedly not declared `Stable`, and thus cannot be used as field type on a `AtomicFlexotask` class.

8.5 Transactional Methods for Multi-Processors

As described in Sec. 5, the prototype implementation of Reflexes is implemented on top of the Ovm virtual machine. Supporting transactional methods in Flexotask involve two non-trivial challenges, both relating to the target platform, the IBM WebSphere Real-Time VM. Unlike the IBM WebSphere Real-Time VM, Ovm (being a research virtual machine) provides built-in support for transactional methods through its *preemptible atomic regions* feature in the virtual machine as well as in its compiler that performs the actual rewriting of bytecodes. Furthermore, whereas Ovm is designed for uni-processors only, IBM WebSphere Real-Time VM provides support for multi-processors.

Moving from a uni-processor virtual machine to one that supports multi-processors breaks a number of assumptions that together make the previously described approach insufficient. Specifically, on a uni-processor with threading controlled by the virtual machine implementing

preemption by the scheduler is straightforward, as the scheduler immediately can roll back any partially applied transactional change to the state of the task. Consequently, when a time-critical task is released by the scheduler, any ordinary thread that might be executing one of its transactional methods will be preempted effectively and thus not make any mutations in memory hereafter. Running on the IBM WebSphere Real-Time VM this assumption does not hold. Instead, the IBM WebSphere Real-Time VM has a multi-processor design and (usually) maps Java threads to operating system threads. In that type of implementation where threads can run in parallel, it is very difficult to implement a roll-back approach for transactions, perhaps impossible without introducing locking overheads that would substantially perturb execution predictability.

In fact, it turns out that the roll-back approach used in Reflexes would not function in a multi-processor setup. Specifically, the problem of knowing exactly *when* the ordinary Java thread has been preempted such that it makes no more mutations is non-trivial, without using some locking scheme. Not knowing if the ordinary Java thread has effectively been preempted prevents the roll-back of the transaction log as the time-critical task cannot execute before the log has been rolled back and memory reset to the previous state. However, the transaction log cannot be rolled back before it has been established that no more mutations are being made (by the ordinary Java thread).

Consequently, Flexotask uses roll-forward approach in which a transactional method defers all memory mutations in a local transaction log until commit time. Having reached commit time, it is checked whether the state of the task has changed during the method invocation, and if so throws an `AtomicException`. The entries in transaction log can safely be discarded in constant time as the mutations will not be applied. If not, the method is permitted to commit its changes efficiently, with the scheduler briefly locked out.

Lacking the preemptible atomic regions feature and to provide support for roll-forward transaction logs, the Flexotask system uses a combination of program transformations and minimal native extensions in the virtual machine to enable method invocations with transactional semantics for a virtual machine with multi-processor support.

8.5.1 Transformation Principles

The principle of the program transformations is to provide transactional semantics to the methods on the Flexotask that are invoked by the ordinary Java thread. Classes not reachable through the call-graph are not affected by the program transformations, and thus the semantics of the rest of the application code should remain unchanged.

As mentioned, the actual program transformations are performed by the Flexotask development tool support, following the initial standard Java compilation and subsequent successful static checking of the Flexotask code against the static safety checks. The program transformations are performed directly on the Java bytecode, rather than in the Java source code, and are thus completely transparent to the programmer. The program transformations involves four parts; call-graph privatization, transactionalizing field operations, wrapping of outermost transactional

methods, and guard class generation.

8.5.2 Call-Graph Privatization

The purpose of call-graph privatization is to generate a transactional variant of the call-graphs of the transactional methods on the Flexotask task. Thus, following privatization for each call-graph there are two variants; one having transactional semantics and a normal with standard Java semantics. The former is invoked exclusively by the ordinary Java thread through the guard, whereas the latter variant is kept around as the Flexotask might itself invoke some of its own methods in the call-graph, and those should not be invoked with transactional semantics as they will always succeed and thus never roll-back.

The starting point of the privatization transformation is the transactional methods declared on the `ExternalMethods` subinterface and implemented by the `AtomicFlexotask` subclass, as seen in Fig. 8.5(a). Starting from here, the call-graph(s) of those methods are traversed method-by-method using Rapid Type Analysis [BS96] on the method bodies. For each reachable method, a duplicate, synthetic method is generated and inserted into the same class where the method originated. Fig. 8.7(c) illustrates this by having two variants of the methods `write` and `update` following privatization, compared to Fig. 8.7(b) before the transformation.

Inevitably, such duplication of methods leads to an increase in the sizes of the affected class files. However, in practice the size increase typically is rather insignificant. The constant pool with its string literals describing classes, fields, types etc. contributes to a significant part of the size of a class file. Duplicating and privatizing a method would, however, for the most part rely on a reuse of existing entries in the constant pool, and apart from the duplication of the method body bytecodes therefore not add significantly to the class file size.

The next step taken during privatization is to ensure that the two variants of each call-graph are kept distinct such that a transactional method cannot invoke a non-transactional, or contrary. This is ensured by adjusting any method invocations performed within the body of a privatized method to invoke the privatized variant of that method. By default synthesized methods inherit the exact same method signature as their original counterparts. This poses a problem since the synthesized methods live in the same class as their original counterparts, and a class cannot have two methods with exactly the same signature (see, [GJSB00] §8.4.2). To encounter this, the synthesized methods are made distinct (from their original counterparts) by appending a dummy parameter of a unique internal type `Privatized` to their signatures, as illustrated in Fig. 8.7(c). Consequently, any method invocations occurring within synthesized methods must be adjusted accordingly.

For the sake of simplicity in the transformation, the dummy parameter is appended to the end of the existing list of parameters in the method signature. To adjust the method invocations accordingly, the bytecodes making up the invocation of the synthesized methods are adjusted by adding an additional parameter to the invocation call. For the actual value being provided to this method invocation, a `null`-argument is simply provided as it is never used. Hence, in

```
public interface HighFreqReaderIntf extends ExternalMethods {
  public void write(int value) throws AtomicException;
}
```

(a)

```
public class HighFreqReader
  extends AtomicFlexotask
  implements HighFreqReaderIntf {

  public void write(int value)
    throws AtomicException {
    ...
    update(value);
  }

  private void update(int value) {
    ...
  }

}
```

(b)

```
public class HighFreqReader
  extends AtomicFlexotask
  implements HighFreqReaderIntf {

  public void write(int value)
    throws AtomicException {
    ...
    update(value);
  }

  public void write(int value, Privatized _)
    throws AtomicException {
    ...
    update(value, null);
  }

  private void update(int value) {
    ...
  }

  private void update(int value, Privatized _) {
    ...
  }
}
```

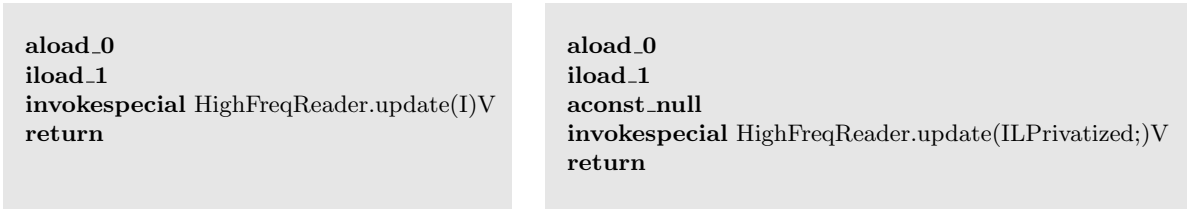
(c)

Figure 8.5: Illustration of privatization of an `AtomicFlexotask` class, expressed in Java source code. (a) shows the Java source code of the `ExternalMethods` subinterface declaring the transactional method to be implemented by the `HighFreqReader`, (b) shows the Java source code of the `HighFreqReader` implementing the transactional method before privatization, and (c) shows the effects of privatizing the transactional method. Note, only for illustration purposes are the effects of privatization expressed using Java source code. In reality, the actual transformations are performed directly on the bytecodes.

the bytecodes an `aconst_null` instruction is simply inserted just before the actual `invoke*`[2] instruction. Fig. 8.6 shows the bytecode of the invocation of the method `update(value)`, as seen in Fig. 8.5, before and after the transformation.

[2]The instruction `invoke*` is used to denote the four invocation instructions of the Java bytecodes: `invokespecial`, `invokevirtual`, `invokeinterface`, `invokestatic`.

This leads to an interesting case where a rewritten method invocation appears as if it could match existing methods having different, yet matching, signatures. Continuing the example in Fig. 8.5, this could happen if the `HighFreqReader` class, seen in Fig. 8.7(c), contained an existing method, say, with the signature `update(int, Ljava/lang/Object)V;`. One might wonder which method would be invoked by an invocation `update(1, null)` occurring in the body of an existing method. This is a case of operator overloading. However, Java resolves operator overloading statically, and thus, at the time of the bytecode rewriting, any overloading issues for the un-rewritten has already been resolved by the standard Java compiler. What remains is thus simply to ensure that the overloading issues of the synthesized methods are resolved correctly.



(a) (b)

Figure 8.6: Transformation of method invocations within privatized methods; redirecting the invocation to privatized methods. (a) shows the bytecode of the method invocation before the transformation, (b) shows the bytecode of the method invocation after the transformation, where the invocation is redirected to the privatized method with its additional method parameter.

8.5.3 Transactionalizing Field Operations

The purpose of transactionalizing field operations taking place within a privatized method is to redirect all field operations involving stable types to a transaction log, such that their effects are deferred until commit time, rather than having them performed immediately in memory. Here field operations include `getfield` and `putfield` bytecode instructions and all primitive and reference-type array load and store bytecode instructions.[3] Note, that in our current implementation we actually transactionalize all field operations, stable or not. The consequence of this missing optimization is a negligible run-time overhead caused by the fact that the number of entries in the transaction log will be a bit larger as it will include mutations made to fields of transient type too. Thus, the time to commit a transaction, i.e., roll-forward the entries in the transaction log, will be a bit longer.

In addition, we do not transactionalize any writes to static variables within a privatized method. This choice is deliberate. The purpose of the transactional methods is to provide non-blocking access to shared data between the ordinary Java thread and the time-critical task. Since the state of the time-critical task consists only of stable objects allocated in the stable heap of the

[3]Specifically, the following array instructions are included: `aaload`, `baload`, `caload`, `daload`, `faload`, `iaload`, `laload`, `saload`, `aastore`, `bastore`, `castore`, `dastore`, `fastore`, `iastore`, `lastore`, `sastore`.

task, any shared state must relate to these objects. Hence, the transactionalization of the field operations exclude writes to static variables.

This redirection of the field operations takes place as a subtransformation of the privatization of the method. Specifically, upon detecting a field operation in the method body, i.e., a `getfield/putfield` instruction, the bytecode instruction is replaced with an `invokestatic` instruction invoking an internal, type-specific native method on the virtual machine. Specifically, the IBM WebSphere Real-Time VM has been extended with a class `TransactionalOperations` to provide a native method for each primitive type, for each array of primitive type, as well as a method for reference types and arrays of reference types.

```
public void write(int value) throws AtomicException {
  state.current.value = value;
}
```

(a)

```
aload_0        // 'this' object
getfield LState; HighFreqReader.state
getfield LChannel; State.current
iload_1        // value to be assigned to the Channel.value field
putfield I Channel.value
```

(b)

```
aload_0        // 'this' object
iconst_0       // index of field 'HighFreqReader.state'
invokestatic com/ibm/realtime/flexotask/TransactionalOperations
                   .getReferenceField(Ljava/lang/Object;I)Ljava/lang/Object;
iconst_1       // index of field 'highfreqread/txn/State.current'
invokestatic com/ibm/realtime/flexotask/TransactionalOperations
                   .getReferenceField(Ljava/lang/Object;I)Ljava/lang/Object;
iload_1        // value to be assigned to field 'Channel.value'
iconst_2       // index of field 'Channel.value'
invokestatic com/ibm/realtime/flexotask/TransactionalOperations
                   .setIntField(Ljava/lang/Object;II)V
```

(c)

Figure 8.7: Transactionalizing field operations in a transactional method. (a) shows the Java source code of the transactional method, (b) shows the original bytecodes of the method body, and (c) shows the bytecodes of the method body having applied the subtransformation. Note how after bytecode rewriting each field is referenced through a numeric index value, rather than through the constant pool index.

Each bytecode instruction representing a field operation holds an index to an entry in the constant pool of the class where the instruction takes place, uniquely identifying the actual field to be manipulated. When redirecting the field operations to the transaction log, it is necessary to maintain such unique identification of each individual field. However, using the constant pool entry index is problematic since the scope of a transaction may cover several transactional methods including several classes, thus introducing the risk of clashes of field indices. Instead, during the rewriting of these field operations, the bytecode rewriter assigns unique indices to each reachable instance field in all the classes of the call-graph of the transaction. This index has to be provided as an argument to the invocation of the native methods in order to identify the fields in the transaction log. Fig. 8.7 provides an example of transactionalizing the field operations of a transactional method.

As can be seen in Fig. 8.7, the reference type `getfield` instructions are transformed into a `invokestatic` instruction on the `TransactionalOperations` class that as arguments takes the object on which the field read takes place, and an index uniquely identifying the field to be read. The same is true for the primitive type `putfield` instruction, though here the value to be assigned to the field is provided as an additional argument.

8.5.4 Wrapping Outermost Transactional Methods

The outermost transactional methods are those synthesized, transactional methods on the `AtomicFlexotask` class that will be invoked indirectly by the ordinary Java thread, i.e., the entry points of the ordinary Java thread to the `AtomicFlexotask` class.

Being the entry points to the `AtomicFlexotask`, these methods must be treated specially. Particularly, these entry point methods have to deal with the fact that the invocation by the ordinary Java thread at this point will cross the boundary between the public heap and the memory area of the `AtomicFlexotask` instance. Consequently, the method bodies of these outermost transactional methods on the `AtomicFlexotask` are wrapped by the rewriter with a *prolog* and an *epilog*. Wrapping these method bodies serves a number of purposes.

Upon entering the transactional method, the first thing occuring is an invocation of the native `performProlog` method on the `AtomicFlexotask` class occurs, which will cause for the transaction log to be prepared and the memory area of the ordinary Java thread to be switched. When an ordinary Java thread invokes transactional methods, its memory context is the public heap. However, the body of a transactional method has to be invoked in special allocation context in which transient objects go to the public heap but the stable ones go to the stable heap of the Flexotask. Thus, the memory context has to be switched before executing the actual instructions of the method body. Note how unlike in Reflexes, we do not pin any reference type arguments (i.e., primitive arrays) that might be passed in to the transactional method.

Following the execution of the actual method body, the epilog will attempt to commit the transaction by invoking the native `performEpilog` method. The epilog checks whether the state of the task has changed out during the method invocation, and if so causes for an `AtomicException` to be thrown back to the ordinary Java thread. If not, the method is permitted to commit its

changes, with the scheduler briefly being locked out. Having committed or aborted the transaction, the epilog then resets the memory context back to the public heap. The `performProlog` and `performEpilog` methods both represent an extension to the virtual machine, and are declared as part of the internal functionality on the `AtomicFlexotask` class, as seen in Fig. 7.6.

Locking out the scheduler during the commit phase of the `performEpilog` method makes it vulnerable to being blocked indirectly by the garbage collector, because the committing thread is an ordinary one that can be paused by the collector while holding the lock. We, thus, require the thread to complete its commit and release the lock before yielding to the collector. Note, that while on a uniprocessor machine transactional methods can be implemented in a lock-free fashion, on a multi-processor machine this methodology does not work. In fact, in our implementation we actually rely on a lock-based approach with priority inheritance. The implications of this scheme is that on a multi-processor machine, in order to support transactional methods, on a lower level, we are subject to some of the deficiencies of lock-based schemes that transactional methods should help us circumvent. However, we have strived for locking out the scheduler in a minimal lockout window, thereby minimizing any negative impact on performance.

Finally, the epilog is also responsible for exception handling. More specifically, the epilog must ensure that the invocation, in the event of an exception, does not escape the outermost transactional method without having reset the memory context back to the public heap. The epilog must handle three types of exceptions; user exceptions, if any, thrown by the outermost reachable method, internal exceptions representing transaction aborts if the restricted thread is released by the scheduler, and finally any unchecked exception that might otherwise occur.

If a user exception is thrown within a transactional method, the object is created with normal Java semantics. By default the exception object and its stack trace are created in transient memory. For the exception object to escape from a call to a transactional method from an ordinary Java thread, the exception object has to be deep-cloned on to the heap before being rethrown. In contrast, recall, in Reflexes we simply just relied on standard RTSJ exception handling whereby the invoking ordinary Java thread would receive a `ThrowBoundaryError`.

Fig. 8.8 illustrates how the Java code skeleton version of the outermost transactional method seen in Fig. 8.7(a) after having been wrapped with a prolog and epilog. The figure does not show the transformed method body of the original method.

8.5.5 Guard Class Generation

At run-time, the rewritten `AtomicFlexotask` class will reside inside its own private memory area, which the ordinary Java thread is given access to through the guard that delegates all invocations. The guard is an instance of a synthesized, anonymous class that implements the same set of `ExternalMethods` marker interfaces as the `AtomicFlexotask` subclass implements, and is transparently provided to the time-oblivious Java code as a delegator for the actual `AtomicFlexotask`.

Whereas the other program transformations are performed on existing classes, the guard class is generated from scratch. Fig. 8.9 shows an example of a generated guard class for the `Atomic-`

```
1
2  public void write(int value, Privatized _)
3    throws AtomicException, UserException {
4    Throwable t = null;
5    try {
6      performProlog();
7
8      // here comes the transformed method body of the original method
9    }
10   catch (Throwable e) {
11     t = e;
12   }
13   finally {
14     if (t instanceof ExecutionAbortedException) // 'performProlog' failed;
15       throw t;                          // memory area could not be switched
16     try {
17       t = performEpilog(t);
18     }
19     catch (ExecutionAbortedException e) { // 'performEpilog' failed;
20       throw new AtomicException(e);       // transaction could not commit
21     }
22     if (t != null) {
23       if (t instanceof UserException) // handling of user exception types
24         throw t; // user code threw an expected exception type
25       if (t instanceof Error)
26         throw t;
27       if (t instanceof run-timeException)
28         throw t;
29       throw new IllegalStateException(t);
30     }
31   }
32 }
33
```

Figure 8.8: Illustration of the effects of wrapping the method body of an outermost transactional method with a prolog and epilog. In lines 4-6, the inserted prolog code can be seen, and in lines 9-31 the epilog code. The transformed method body of the outermost transactional method is inserted at the placeholder of line 8 as indicated by the code comment. Note, only for illustration purposes are the effects of wrapping expressed using Java source code. In reality, the actual wrapping subtransformation is performed directly on the bytecodes.

`Flexotask` subclass seen in Fig. 8.7(c). Note, the package and class name of the generated guard class is equivalent to the `AtomicFlexotask` subclass that it delegates to, though a `$Guard` suffix is added to avoid name clashes. The guard class must implement all the `ExternalMethods` subinterfaces of the `AtomicFlexotask` subclass that it delegates to. Furthermore, for each of the methods declared in these subinterfaces, a synthesized method with the equivalent method signature is generated on the guard class, with a method body that simply relays the invocation to the `AtomicFlexotask` instance that it holds a reference to.

To associate the guard with the actual `AtomicFlexotask` instance, all guard classes must have

```
public class HighFreqReader$Guard implements HighFreqReaderIntf {
  private HighFreqReader flx;

  public void setDelegate(AtomicFlexotask flx) {
    this.flx = (HighFreqReader) flx;
  }

  public void write(int value) throws AtomicException {
    flx.write(value, null);
  }
}
```

Figure 8.9: Illustration of the automatically generated guard class, delegating for an `AtomicFlexotask` class. Note, only for illustration purposes is the guard class expressed using Java source code. In reality, the guard class is generated directly in bytecode format.

a default `setDelegate` method expecting an argument of type `AtomicFlexotask`, as seen in Fig. 8.9. This method is invoked by the Flexotask run-time engine during graph initialization for setting the actual `AtomicFlexotask` subclass instance on the guard. When the Flexotask graph terminates, the `setDelegate` is invoked again by the Flexotask run-time engine with a `null` value to ensure that the guard does not keep alive a no-longer living task.

8.6 Transaction Log

The transaction log uses a roll-forward approach where all memory operations are redirected and performed in a transaction log rather than on the actual memory addresses. During a transaction, when a field in an object is being assigned, an entry representing that field in that particular object is added to the transaction log. The entry points to the address in memory holding the latest value that eventually should be assigned to the instance field upon commit. If an entry representing that field is already in the log, the entry is changed to point to the memory address of the new value. Likewise, when a field is being read, it is first sought for in linear time in the transaction log. If found, the value at the memory address pointed to by the log entry is returned. If no entry in the log represents the field, it means that the field was never assigned during the current transaction. In that case, the field value is read directly from the memory address of the field in the object. Only if the transaction commits will the memory operations represented by the entries in the transaction log be performed directly in memory.

The actual transaction log is implemented in the native layer of the virtual machine as a unidirectional linked list. In our current implementation the memory consumption of the transaction log is unshrinkable, meaning that memory allocated for entries in the transaction log are never freed again. Instead, upon resetting the transaction log, these entries are reset and put on a free-list for reuse for subsequent transactions. Only if the free-list runs out of available log entries are new allocated.

9 Empirical Evaluation

This chapter describes a number of empirical evaluations of our integrated Flexotask implementation. Like with Reflexes, we evaluate Flexotask on predictability and performance using two applications, but this time a high frequency reader and an avionics collision detector. We also report on the execution time for our static analysis at compile- and initialization-time for both applications.

9.1 Methodology

The experiments were conducted using the IBM WebSphere Real-Time VM extended with support for Flexotask. The virtual machine includes support for high resolution timing on real-time kernels, a real-time garbage collector [BCR03], and an implementation of RTSJ. The IBM WebSphere Real-Time VM also includes experimental features added for Eventrons, providing object pinning/unpinning and the ability to exempt certain threads from being paused by the global heap garbage collector. The Flexotask implementation extends the Exotasks implementation [ABI+07, IBM], which provides private per-task heaps, and deep copying between heaps. In addition, we added support for transient allocation and roll-forward transactions.

As our execution platform, we used an IBM blade server with 4 dual-core AMD Opteron 64 2.4 GHz processors and 12GB of physical memory. The operating system used was Linux (kernel version 2.6.21.4 in the RHEL 5 real-time configuration).

9.2 Predictability

We evaluate predictability of a *high frequency reader* benchmark (540 lines of code). The application has an atomic Flexotask scheduled at a period of 100 μs. At each periodic execution, the task reads available data on its input buffer in circular fashion into its stable state. An ordinary Java thread that runs continuously feeds the Flexotask with data on its input buffer by invoking a transactional method on the task approximately every 20 ms. Out of 3,000 invocations of the transactional method, 516 of them aborted, indicating that the atomic Flexotask feature was being heavily exercised. To evaluate the influence of computational noise and garbage collection, another ordinary Java thread runs concurrently, continuously allocating at the rate of 2MB per second, using 48 byte objects and maintaining a live set of 40,000 objects. To avoid perturbations caused by the JIT-compiler, we ran this test in interpreted mode.

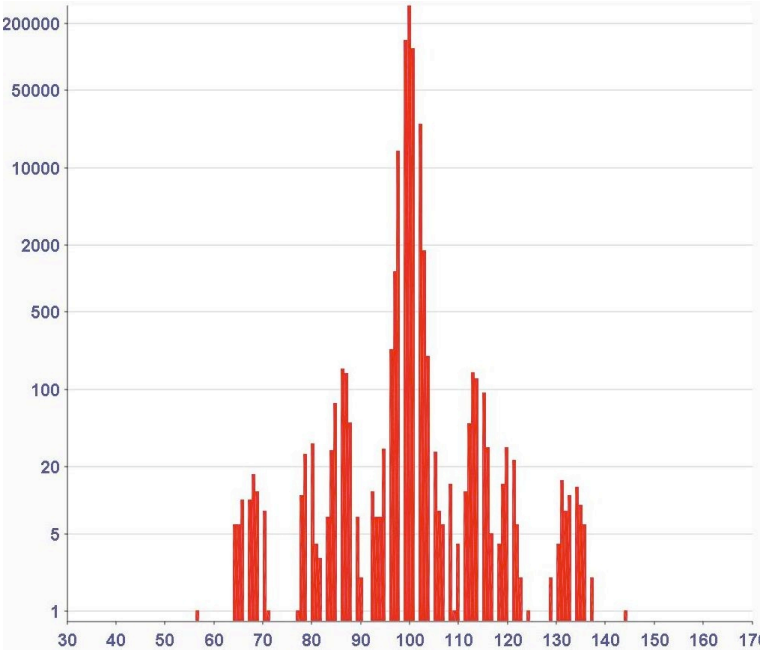


Figure 9.1: Frequencies of inter-arrival times of an atomic Flexotask scheduled with a period of 100 μs, executing concurrently with (1) an ordinary Java thread communicating by transactional invocations, and (2) an ordinary Java thread simulating regular memory consumption by continuously allocating at 2MB per second. The x-axis depicts the inter-arrival time of two consecutive executions in microseconds. The y-axis depicts the logarithm of the frequency.

Fig. 9.1 shows a histogram of the frequencies of inter-arrival times of the periodically scheduled atomic Flexotask, i.e., the time between two consecutive executions. The figure contains observations covering almost 600,000 periodic executions. As can be seen in the figure, all observations of the inter-arrival time are centered around the scheduled period of 100 μs. Overall, there are only a few microseconds of jitter to be seen in the figure, with inter-arrival times ranging from 57 to 144 μs.

9.3 Performance

To evaluate performance, we considered a larger application in the form of the avionics collision detector described in Sec. 7.7. The collision detector (30,000 lines of code) consists of three threads running concurrently: the `DetectorTask` running at a period of 20 ms, the simulator thread generating flight data and communicating with `DetectorTask` every 20 ms, and the 2MB per second allocator thread described in the previous section. Because the 20 ms period allows more slack than in the predictability test (necessary both for realism and to allow the simulator to keep up), we instructed the allocator thread to keep 150,000 objects live in order to cause more garbage collection overhead and a greater opportunity for conflict.

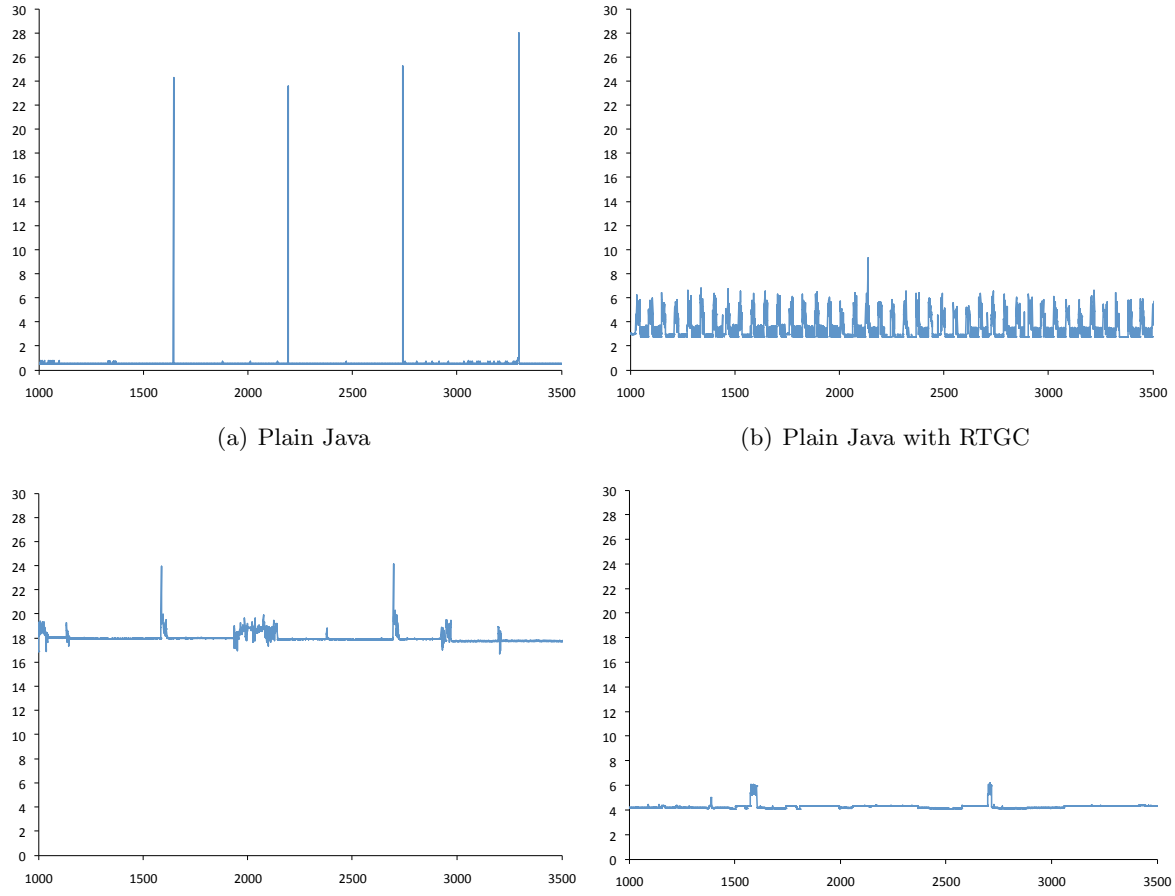
To have a baseline with which to compare our measurements, we implemented two additional variants of the collision detector, respectively a plain Java variant where the time critical thread was just an ordinary thread, and a RTSJ variant making use of scoped memory areas. The plain Java version was run both under a normal garbage collector (the non-real-time IBM J9 collector with default parameters), and under the real-time garbage collector of the IBM WebSphere Real-Time VM. For this experiment, we measured performance as time taken by the detector to process a frame.

Fig. 9.2 shows the results of our measurements. For the plain Java variant with a non-realtime collector Fig. 9.2(a), the worst-case observed processing time over the entire run was around 28 ms, which is not surprising given that the virtual machine uses an ordinary non-real-time garbage collector. With the real-time garbage collector, this number declines substantially but this comes at the expense of mutator utilization, thereby increasing the average processing time. The smaller but non-negligible jitter in Fig. 9.2(b) happens because a varying number of the garbage collector's work quanta can fall within the relatively long processing times (about 4 ms) for the detector.

Both the RTSJ variant Fig. 9.2(c) and Flexotask Fig. 9.2(d) are largely impervious to such interference. We note that they are not entirely impervious: each has two spikes that correspond to garbage collections though other garbage collections in the run pass without incident. In the Flexotask version Fig. 9.2(d), the two spikes are due to a thread being preempted while holding a lock needed by the scheduler, a known problem relating to the problem mentioned earlier that an ordinary Java thread can be paused by the garbage collector during its commit. We will address this small implementation problem as future work. In the RTSJ version the spikes are unexplained but probably represent a flaw in the implementation of the virtual machine.

Clearly, the best-case performance time of the RTSJ variant is significantly slower than that of any other variant. We attribute the slowdown to the dynamic checks imposed during run-time by the virtual machine to ensure safety of pointer manipulation when using the RTSJ scoped memory areas.

In summary, we have shown how four different variants of our collision detector application are subject to a varying degree of interference caused by the presence of garbage collection. As expected, the plain Java variant experiences infrequent, but large, latencies when running on a non-real-time collector and much smaller latencies but still noticeable jitter when running on



(a) Plain Java

(b) Plain Java with RTGC

(c) RTSJ with Scoped Memory

(d) Flexotask

Figure 9.2: Comparing performance of four different variants of the collision detector benchmark. The x-axes show the data frames processed, numbering from 1 (only a representative set of frames are shown), and the y-axes the processing time in milliseconds for the individual frame.

the real-time collector. The RTSJ variant has low jitter but poor average-case performance. In contrast to these, the Flexotask variant runs at high performance with the smallest amount of jitter.

9.4 Static Analysis Performance

We measured the time needed for validating the code of our two benchmarks applications at compile nd initialization time. Whereas initialization time validation was performed on the platform described above, compile time validation occurred on a development machine running JDK 1.5.0_07-87 on a Intel Core Duo, 2.16GHz with 2GB of physical memory.

	High Frequency Reader	Collision Detector
Compile-time Validation	33 *ms*	173 *ms*
Bytecode Rewriting	10 *ms*	51 *ms*
Initialization-time Validation	332 *ms*	699 *ms*

Table 9.1: Static analysis times for the two benchmark applications.

Tab. 9.4 shows the empirical measurements of the time to perform the various stages of the code analysis. As can be seen, it takes twice as long to validate the collision detector. This is not surprising given the difference in code size. The longer time taken at initialization time primarily reflects the fact that "validation" actually includes the time to instantiate and schedule the Flexotask graph in addition to simple checking. Also, the checking is more detailed since it is done in a data-sensitive fashion.

9.5 Software Engineering Aspects

We briefly comment on our experience refactoring the collision detector application to use Flexotask APIs. The collision detector code obtained from [ZNV04] consisted of 195 files, containing 241 classes (with around 30,000 lines of code), and employed RTSJ APIs. Converting it to Flexotask required modification of 8 files and adding `Stable` declarations to 7 classes. The main changes were in the setup portion of the application; the original version had code for creating RTSJ-style real-time threads, whereas the Flexotask version created a single-node graph.

The other main change was in the communication between ordinary Java threads and the real-time task. In order to pass the validation phase, we had to ensure that objects shared between the two were reference-immutable. Finally, in a number of places where the RTSJ code had to resort to reflective invocation, the calls were transformed into normal allocations of stable classes in the Flexotask version. The effort going from the earlier version of the code was modest and had the side-effect of making the code easier to understand.

Part V

Conclusion

10 Conclusion

This thesis presented *Reflexes*, a simple, statically type-safe programming model that makes it easy write and integrate simple periodic tasks or complex stream processors, both observing real-time timing constraints in the sub-millisecond range, into larger Java applications running on the same Java virtual machine. Reflexes provide means for type-safe, obstruction-free interaction between time-critical tasks and time-oblivious Java code. The Reflex programming model is non-intrusive in that it does not require changes to the standard Java syntax, nor does it require refactoring of existing code, permitting a high degree of reuse of standard libraries and legacy code.

In this thesis, we set out to find the answers to a number of questions as described in the Problem Statement. Below, we will summarize the conclusions of our findings.

Circumventing Garbage Collection Interference

- **Question:** *How can the latency introduced by garbage collection be circumvented such that sub-millisecond predictability is not compromised, yet avoiding the deficiencies of the* `NoHeapRealtimeThread`*?*

We presented the Reflex restricted programming model in which the unit of restriction and execution is a *task*. Like in other restricted programming models, Reflex tasks execute free of garbage collection interference by (1) executing in a separate private memory area that is not reachable from the public heap garbage collector, and (2) being executed by real-time threads running at higher priority than any other thread, including the garbage collector. The result of these two features is that Reflex tasks at any point in time can preempt the garbage collector and thereby not be subject to garbage collection-related latencies.

Through a number of empirical experiments, both specific benchmark applications as well as real-world applications, we documented that with Reflexes time-critical tasks achieve a very high level of predictability in their execution, and that this occurs even at execution periods below a millisecond.

Ensuring Type-Safety

- *How can the run-time checks maintaining the memory region integrity be avoided without compromising type-safety?*

As we described and pointed out, the necessity for performing checks concerning memory region integrity comes with the introduction of code being executed in different memory contexts, and in particular, when references cross the boundaries of these contexts. Wanting to avoid applying expensive run-time checks as with the `NoHeapRealtimeThread` to enforce memory region integrity, we presented a set of safety checks that can be statically enforced and reasoned about their correctness. We described how the static safety checks work by restricting the expressive power of Reflex tasks, thereby ensuring that the tasks will never perform operations that could cause the time-critical tasks nor the time-oblivious Java code to experience a dangling pointer or observe a heap-allocated object in an inconsistent state.

While the static safety checks are strict enough to ensure type safety for time-critical tasks, they are non-intrusive in that they allow for reuse of most standard library classes and legacy code, as we investigated by subjecting the Java collection framework to the set of restrictions. Furthermore, as we described, the scope of these checks concerns only the time-critical parts of the code, not the remainder of the Java application, again promoting reuse and integration into existing application code.

We also described and illustrated how, even having enforced the restrictions, it is necessary to extend the virtual machine with the ability to enable pinning of certain objects on the heap as they are referenced from objects in different memory contexts not reachable by the public heap garbage collector. Concretely, in Reflexes, we identified heap-allocated, reference type static variables and instance objects provided as transactional methods argument to be pinned.

In our integration of Reflexes together with the Eventrons and Exotasks models from IBM Research into a unified, restricted programming model, *Flexotask*, we realized and presented a relaxation of the set of static safety checks, as a direct consequence of the introduction of a guard object, shielding and delegating to the time-critical task. Furthermore, relating to transactional methods, through an evolutionary change of the allocation context of transient objects from a dedicated transient area to the public heap, we could avoid the necessity for pinning of objects – more specifically, we described how with this change, we no longer needed to pin heap-allocated, reference type arguments provided to transactional methods.

Enabling Type-Safe, Non-Blocking Communication

- *How can threads observing sub-millisecond temporal requirements interact with time-oblivious threads in a type-safe manner without sacrificing predictability, and without requiring extensive modification of legacy code?*

We described the challenges involved in enabling communication between time-critical tasks and time-oblivious code, and presented the two means of communication possible in the Reflex programming model. We described how to safely communicate using static variables, by restricting their types to those that are primitive or reference-immutable. We then introduced an obstruction-free, transactional communication scheme based on special methods with transactional semantics to be invoked by ordinary Java threads, and explained why it is required to restrict parameters on such methods to those of primitive or primitive array types. Finally, we discussed when to use which type of communication, and also proposed when to use a combination of the two.

We described our initial prototype implementation of Reflexes implemented on top of an experimental real-time Java virtual machine with a uni-processor design, and providing low-level mechanisms for achieving transactional semantics by enabling memory mutations to roll-back. We presented experimental results of running mixed mode applications, i.e., applications where time-oblivious threads interact with time-critical tasks, of our prototype implementation, and showed that transactional methods indeed enable obstruction-freedom for the time-critical task. Specifically, our results demonstrate that time-critical tasks can achieve a high degree of predictability, even when running at high frequencies, while an ordinary Java thread invokes transactional methods on it.

For our implementation of the unified Flexotask programming model, we described how we used a standard industrial-strength real-time Java virtual machine with a multi-processor design that does not provide low-level support for transactional methods. Hence, we presented a set of program transformations and necessary native support in the virtual machine required in order to achieve transactional semantics of method invocations, and described a new approach based on a roll-forward transaction log in order for transactional methods to work correctly on a multi-processor virtual machine. Also for this implementation we presented encouraging evaluation results demonstrating the viability of transactional methods and the programming model in general.

10.1 Contributions

Having described the conclusions of our findings to the questions posed in the Problem Statement, we now summarize the contributions of this thesis:

- **Programming Model** – We presented the design of a simple, type-safe restricted programming model, *Reflexes*, facilitating the construction of highly responsive applications

in Java. The Reflex programming model makes it easy write and integrate simple periodic tasks or complex stream processors, both observing real-time timing constraints in the sub-millisecond range, into larger time-oblivious Java applications.

- **Static Safety Checks** – Standard real-time programming constructs for circumventing interference from garbage collection rely on the enforcement of type restrictions at run-time; an approach that adds a significant run-time overhead relating to safety of memory operations. To avoid the need to apply expensive checks during run-time, we described an informal specification of a set of static safety checks inspired by ownership types for statically ensuring the safety of memory operations within a Reflex task while at the same time permitting communication with time-oblivious code. In particular, the static safety checks propose a novel notion of *implicit* ownership, rendering superfluous the need to declare ownership parameters on class declarations. Furthermore, the static safety checks are non-intrusive by permitting unmodified reuse of legacy code with no requirement to rewrite standard libraries.

- **Obstruction-free, Transactional Communication Scheme** – To facilitate non-blocking communication between time-critical tasks and time-oblivious Java code, we described a scheme based on obstruction-free, transactional communication, ensuring that the time-critical Reflex task will not violate its temporal requirements following interaction with time-oblivious code. Furthermore, the thesis presented a general design of the communication scheme enabling implementations on multi-processors, where it cannot be assumed that the release of the time-critical task causes for the immediate execution halt of a concurrently running time-oblivious thread.

- **Implementation and Integration** – We described two implementations of the Reflex programming model. To demonstrate viability of the approach, we developed a stand-alone prototype implementation of Reflexes on a research virtual machine that comes with a uni-processor design, and native support for transactional methods. In addition, we added extensions to the `javac` compiler to support the static safety checks.

  To provide a more powerful and flexible programming model, we integrated the Reflex programming model with two existing restricted programming models from IBM Research into a unified programming model, *Flexotask*. To demonstrate the strength and flexibility of our approach, we implemented Flexotask on a commercial virtual machine with multi-processor support. Furthermore, we integrated development tool support for Flexotask into the Eclipse IDE, among other features the static safety checking.

- **Empirical Evaluation** – For both our implementations, we provided a number of empirical evaluations of the ability of the programming model to enable the development of real-time applications providing (a) sub-millisecond response times with a high degree of precision, and (b) throughput better or comparable to equivalent application variants built for alternative approaches. In both cases we demonstrated encouraging results through benchmark applications running simple periodic tasks in isolation to demonstrate the baseline performance, as well as concurrently with communicating, ordinary Java threads to demonstrate predictability and performance when interacting.

10.2 Open Problems

In our pursuit of the answers to the questions posed in the Problem Statement, we have come across a number of loose ends and challenges that are very intriguing:

- **Evaluating the possibility of reference-immutable objects as transactional methods arguments.** As described earlier in the thesis, allowing for any reference-immutable type to be used as transactional method argument imposes two problems; statically determining the type safety, and during run-time the cost of constantly pinning and unpinning a potentially large reference-immutable data structure for each method invocation. The latter challenge disappears when changing allocation context to the public heap, and pinning is then no longer necessary. The challenges involved in statically determining the type safety of using reference-immutable, transient types has already been described earlier and remains clear. However, whereas at startup time shared instance objects and static variables can be analyzed based on objects, the actual types of the transactional methods arguments are not known before invocation time, and must thus be analyzed based on a class-based level – even at startup time. How well this would work in practice, we would like to try out.
- **Validating safety of rewritten, transactional methods.** In our current version of our Flexotask implementation, the development time checker applies a number of program transformations, described in Sec. 8.5, to facilitate transactional semantics on the transactional methods. In our current initialization time checker, these program transformations are blindly trusted and thus not checked to adhere to the safety requirements. This poses a potential security problem in that these methods could be counterfeited to compromise type safety. Thus, interesting work in this area include extending our initialization time checker with the ability to not only check the method body of the rewritten classes according to the static safety checks, but also to perform verification that the rewritten outermost transactional methods indeed always execute their epilogs and that they perform all the mutations in the transaction log.
- **Scheduler support for non-periodic executions.** The scheduling in the current version of both Reflexes and Flexotask is based on the concept of a period, hence the time-triggered scheduler. It could, however, be interesting to explore alternative, non-periodic scheduling policies, e.g., event-based scheduling tied to low level I/O like reading from a network interface.
- **A stream-based scheduler for Flexotask.** In Sec. 8.2.2, we have described a timing grammar and scheduler for supporting complex stream processing applications in Flexotask. We would like to continue this work and perform experimental results comparing the execution on a multi-processor virtual machine with an equivalent execution on a uniprocessor machine in order to learn the cost of parallelizing the application (given thread context switches), and under which circumstances the performance increases outweighs this cost.

Bibliography

[ABC+06] Austin Armbuster, Jason Baker, Antonio Cunei, David Holmes, Chapman Flack, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A Real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems (TECS)*, 2006.

[ABG+08] Joshua Auerbach, David F. Bacon, Rachid Guerraoui, Jesper H. Spring, and Jan Vitek. Flexible Task Graphs: A unified restricted thread programming model for Java. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–11, New York, NY, USA, 2008. ACM.

[ABI+07] Joshua Auerbach, David F. Bacon, Daniel T. Iercan, Christoph M. Kirsch, V. T. Rajan, Harald Roeck, and Rainer Trummer. Java takes flight: time-portable real-time programming with Exotasks. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, volume 42, pages 51–62, New York, NY, USA, 2007. ACM.

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, 1986.

[ANMM06] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 57–74, New York, NY, USA, 2006. ACM.

[ANS] ANSI/ISO. *ISO/IEC 8652:1995 Information Technology – Programming Languages – Ada ISO/IEC 8652:1995/Cor 1:2001 Information Technology – Programming Languages – Ada.*

[Apa] Apache – Byte Code Engineering Library. `http://jakarta.apache.org/bcel/`.

[Arm97] Joe Armstrong. The development of Erlang. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, pages 196–203. ACM Press, 1997.

[BB02] Albert Benveniste and Gérard Berry. *The synchronous approach to reactive and real-time systems.* Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[BCC+03] Greg Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Daniel Dvorak, Brian Giovannoni, Mark Indictor, Kenny Meyer, Alex Murray, and Kirk Reinholtz. 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)*, pages 361–369, 2003.

[BCF+06] Jason Baker, Antonio Cunei, Chapman Flack, Filip Pizlo, Marek Prochazka, Jan Vitek, Austin Armbruster, Edward Pla, and David Holmes. A real-time java virtual machine for avionics - an experience report. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 384–396, Washington, DC, USA, 2006. IEEE Computer Society.

[BCR03] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 285–298, New Orleans, Louisiana, January 2003.

[Ber91] Gerard Berry. A hardware implementation of Pure Esterel. Technical Report 06/91, Sophia-Antipolis, France, 1991.

[BG92] Gerard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th Annual ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA)*, November 2002.

[BN03] Edward G. Benowitz and Albert F. Niessner. Experiences in adopting real-time java for flight-like software. In *Proceedings of the International workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, pages 490–496, 2003.

[BR01] William S. Beebee and Martin C. Rinard. An implementation of scoped memory for real-time java. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT)*, pages 289–305, London, UK, 2001. Springer-Verlag.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th Annual ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA)*, pages 324–341, 1996.

[BSBR03] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2003.

[Bur99] Alan Burns. The ravenscar profile. *ACM SIGAda Ada Letters*, XIX(4):49–52, 1999.

[Car02] Jan Carlson. Languages and methods for specifying real-time systems. Technical report, Mälardalen Real-Time Research Centre, Department of Computer Science and Engineering, Mälardalen University, Sweden, August 2002.

[CC03] Angelo Corsaro and Ron K. Cytron. Efficient memory reference checks for Real-time Java. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2003.

[CIEE94a] Institute of Electrical CORPORATE IEEE and Inc. Staff Electronics Engineers. *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX): System Application Program Interface (API), Amendment 1: Realtime Extension (C Language), IEEE Std 1003.1b-1993.* IEEE Standards Office, New York, NY, USA, 1994.

[CIEE94b] Institute of Electrical CORPORATE IEEE and Inc. Staff Electronics Engineers. *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX): System Application Program Interface (API), IEEE Std 1003.1-1993.* IEEE Standards Office, New York, NY, USA, 1994.

[CIEE95] Institute of Electrical CORPORATE IEEE and Inc. Staff Electronics Engineers. *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX): System Application Program Interface (API), Amendment 2: Threads Extension (C Language), IEEE Std 1003.1c-1995.* IEEE Standards Office, New York, NY, USA, 1995.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 178–188, Munich, West Germany, January 21–23, 1987. ACM SIGACT-SIGPLAN, ACM Press.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Annual ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, pages 48–64. ACM, October 1998.

[Det04] David Detlefs. A hard look at hard real-time garbage collection. In *7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 23–32, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[GB00] James Gosling and Greg Bollella. *The Real-Time Specification for Java.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[GGB87] Thierry Gautier, Paul Le Guernic, and Löic Besnard. SIGNAL: A declarative language for synchronous programming of real-time systems. In *Proceedings of the Functional Programming Languages and Computer Architecture*, pages 257–277, London, UK, 1987. Springer-Verlag.

[GGH+05] Benoit Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper Honig Spring. Frugal Mobile Objects. Technical report, 2005.

[GGH+06] Benoit Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper H. Spring. Frugal Mobile Objects. In *Proceedings of the Euro-American Workshop on Middleware for Sensor Networks, co-located with the 2nd International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2006.

[GGH+07] Benoit Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper H. Spring. Pervasive computing with Frugal Objects. In *Proceedings of the IEEE 21st International Conference on Advanced Information Networking and Applications (AINA)*, Niagara Falls, Canada, May 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[GTK+02] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. *ACM SIGARCH Computer Architecture News*, 30(5):291–303, 2002.

[Hen98] Roger Henriksson. *Scheduling Garbage Colection in Embedded Systems*. PhD thesis, Lund University, July 1998.

[HGB78] Jr. Henry G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[HHK01] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT)*, pages 166–184, London, UK, 2001. Springer-Verlag.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[IBM] IBM Research – Expedited Real-Time Threads. `www.alphaworks.ibm.com/tech/xrts`.

[Jav] Java Community Process – JSR-1: Real-Time Specification for Java. `jcp.org/en/jsr/detail?id=1`.

[KR80] Brian W. Kernighan and Dennis M. Ritchie. The C programming language. *Encyclopedia of Computer Science*, 1980.

[Lee03] E.A. Lee. Overview of the Ptolemy project. Technical Report UCB/ERL M03/25, EECS Department, University of California, Berkeley, 2003.

[LS88] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 260–267, 1988.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[MAE+84] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, Massachusetts, second edition, 1984.

[MBC+05] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time Java. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS)*, December 2005.

[Mica] Microsoft – The .NET Common Language Runtime. `msdn.microsoft.com/en-us/netframework/aa663296.aspx`.

[Micb] Microsoft Switzerland – Sieben zukunftsweisende Projekte für den ICES. `www.microsoft.com/switzerland/mediacorner/de/PressRelease.aspx?id=e69971ae-2cb0-4882-b335-95debfc8d79b#msg`.

[NB03] Albert F. Niessner and Edward G. Benowitz. Rtsj memory areas and their affects on the performance of a flight-like attitude control system. In *Proceedings of the International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, pages 508–519, 2003.

[NVP98] James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, pages 158–185, London, UK, 1998. Springer-Verlag.

[PFHV04] Filip Pizlo, Jason Fox, David Holmes, and Jan Vitek. Real-time Java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, Vienna, Austria, May 2004.

[PV03] Krzysztof Palacz and Jan Vitek. Java subtype tests in real-time. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 378–404, Darmstadt, Germany, July 2003.

[PV06] Filip Pizlo and Jan Vitek. An emprical evaluation of memory management alternatives for Real-Time Java. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS)*, pages 35–46, Washington, DC, USA, 2006. IEEE Computer Society.

[Rep93] John H. Reppy. *Concurrent ML: Design, Application and Semantics*, volume 693 of *Lecture Notes In Computer Science*. Springer-Verlag, 1993.

[RHH85] Jr. Robert H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[RT] High Resolution Timers. `www.tglx.de/projects/hrtimers/2.6.17/`.

[Rui05] J. F. Ruiz. Mission-Critical On-Board Software Using the Ada 95 Ravenscar Profile. In *DASIA 2005 - Data Systems in Aerospace*, volume 602 of *ESA Special Publication*, August 2005.

[SAB+06] Daniel Spoonhower, Joshua Auerbach, David F. Bacon, Perry Cheng, and David Grove. Eventrons: a safe programming construct for high-frequency hard real-time applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 41, pages 283–294, New York, NY, USA, 2006. ACM.

[ScZ05] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[SGV08] Jesper Honig Spring, Rachid Guerraoui, and Jan Vitek. Integrating Hard Real-Time Tasks into Java with Reflexes. In *ACM Transactions on Embedded Computing Systems – Special Issue on Java Technologies for Real-Time and Embedded Systems (JTRES)*, ACM Transactions on Embedded Systems. ACM, ACM, 2008.

[SGVS99] R. Sekar, Y. Guang, S. Verma, and T. Shanbhag. A high-performance network intrusion detection system. In *ACM Conference on Computer and Communications Security*, pages 8–17, 1999.

[Sim] Simulink. `www.mathworks.com/products/simulink`.

[SPGV07a] Jesper H. Spring, Filip Pizlo, Rachid Guerraoui, and Jan Vitek. Reflexes: Abstractions for highly responsive systems. In *Proceedings of the 3rd International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, 2007.

[SPGV07b] Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. StreamFlex: High-throughput stream programming in Java. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA)*, 2007.

[Str91] Bjarne Stroustrup. *The C++ programming language*. Addison Wesley, 1991.

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC)*, April 2002.

[ZBH+08] Tian Zhao, Jason Baker, James Hunt, James Noble, and Jan Vitek. Implicit ownership types for memory management. *Science of Computer Programming*, 71(3):213–241, 2008.

[ZNV04] Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, 2004.

Curriculum Vitae

Jesper Honig Spring, Danish citizen, born November 7th 1972 in Copenhagen of parents Benedikte Birgitte Honig Jensen and Jens Ove Spring.

Joined the Distributed Systems Group of Prof. Eric Jul, Department of Computer Science, University of Copenhagen (DIKU), as well as Copenhagen Business School in autumn 1991 (evening lessons). Completed B.A. in Informatics and Management Accounting from Copenhagen Business School in spring 1998 and in winter 1999 graduated as M.Sc. in Computer Science with grade A.

Hereafter pursued 4 years of industry experience, among others as invited visitor at IBM Almaden Research Center in year 2000, architecting a replicated version of IBM TSpaces based on Master's thesis results, and from 2001 as Senior Software Developer at Softwired AG, Zurich, Switzerland, working on a clustered JMS messaging server. Holds a patent for a state-machine approach to continuing message passing operations while still guaranteeing JMS semantics during partial cluster failures, including network partitioning (USPTO #20030009511).

Commenced Ph.D. in April 2004 in the Distributed Programming Laboratory (LPD) under the supervision of Professor Rachid Guerraoui of the School of Computer and Communication Sciences at École Polytechnique Fédérale de Lausanne, and as of January 2008 also under supervision of Professor Jan Vitek of the Computer Science Department at Purdue University.