DISS. ETH NO. 22101

Correctness and Execution of Concurrent Object-Oriented Programs

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES OF ETH ZURICH
(DR. SC. ETH ZURICH)

presented by
SCOTT GREGORY WEST
Master of Science, McMaster University, Canada

born on
December 5th, 1983

citizen of
Canada

accepted on the recommendation of

Prof. Dr. Bertrand Meyer, examiner
Dr. Sebastian Nanz, co-examiner
Prof. Dr. Jonathan Ostroff, co-examiner
Prof. Dr. Mauro Pezzè, co-examiner

2014

Acknowledgements

This work is the product of long days and nights, and was only made possible by the help and support of numerous people in my life.

First I must thank my parents, Gina and Garry, who have always encouraged me and given me the support and encouragement to pursue my goals. Without them I could have never started this journey, let alone completed it. I also greatly appreciate the wisdom and support of my grandparents, Charles, Mary, and Barbara.

Wolfram Kahl, who first got me started on the idea of continuing in academia and provided a perfect start to the process by supervising my Masters thesis, has my deepest appreciation.

There are numerous people around me who have supported me over the years: Marco P., Benjamin, Marco T., Claudia, Michela, Chris, Georgiana, Carlo, Martin, Max, Cristiano, Jason, Stephan, Mischael, Juri, Jiwon, Andrey, and Alexey.

The relationship I have with my office-mates, Nadia and Christian, is one of the things that I will treasure most from this time. I couldn't have hoped for a more perfect pair of people to work with every day.

Sebastian, who has always been there to advise and ever-so-gently guide me throughout the thesis, deserves much thanks for his patience, hard work, and support over the years.

I would also like to thank Bertrand Meyer, who gave me the chance to come to ETH Zürich. This was a fantastic opportunity and I will always be grateful for the guidance and support he has given (and the occasional European history lesson); he always encouraged me to aim higher, something I will not forget.

I thank the external examiners who took the time to read and evaluate my thesis: Jonathan Ostroff and Mauro Pezzè. Their time and valuable input is greatly appreciated.

Lastly, I must thank my caring and beautiful wife Nancy. She moved to Switzerland to be with me while I studied, provided support and help in the harder times, and reminded me not to be complacent when things were easier. She gave me exactly what I needed, when I needed it, and I will always remember and appreciate that.

Funding. This work was partially supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389, the Hasler Foundation, ETH (ETHIIRA), the Swiss National Foundation, and Microsoft (Multicore award).

Contents

1 Introduction **1**
- 1.1 Approach . . . 2
- 1.2 Contributions . . . 3
- 1.3 Organization . . . 4

2 Background **5**
- 2.1 Object-oriented programming . . . 5
- 2.2 Specifications and contracts . . . 6
- 2.3 Eiffel . . . 7
- 2.4 Threaded model . . . 7
- 2.5 SCOOP model . . . 9
- 2.6 Types of concurrency bugs . . . 12

I Correctness **15**

3 Static deadlock detection **17**
- 3.1 Deadlocks in SCOOP . . . 18
  - 3.1.1 Types and handlers . . . 19
  - 3.1.2 Locking through calls . . . 19
  - 3.1.3 Example . . . 19
- 3.2 Locking operational semantics . . . 20
  - 3.2.1 Syntax . . . 20
  - 3.2.2 Locking semantics . . . 21
- 3.3 Ensuring deadlock freedom . . . 24
  - 3.3.1 Annotation language . . . 25
  - 3.3.2 Well-formed programs . . . 26
  - 3.3.3 Deadlock freedom . . . 29
  - 3.3.4 Reducing annotations . . . 30
  - 3.3.5 Evaluation . . . 31
- 3.4 Tool . . . 32
- 3.5 Coq proof . . . 33
- 3.6 Related work . . . 39
- 3.7 Conclusion . . . 40

4 Testing of programs with contracts **43**
- 4.1 Premise and overview . . . 43
  - 4.1.1 Overview . . . 45
  - 4.1.2 Example . . . 46
- 4.2 Demonic testing . . . 48
  - 4.2.1 Application of rely-guarantee reasoning . . . 48

4.2.2 Class transformation 49
4.2.3 Routine instrumentation 50
4.2.4 Handling synchronization primitives 50
4.3 Applying demonic testing to SCOOP 52
4.3.1 Specializing demonic testing 52
4.3.2 How SCOOP makes it easier 53
4.3.3 Handling wait conditions 54
4.4 DemonL 54
4.4.1 The domain description language 55
4.4.2 Interference as satisfiability 56
4.4.3 The tool 60
4.5 Experimental evaluation 61
4.5.1 Conversion from source programs 61
4.5.2 Results 61
4.5.3 Annotation complexity 63
4.5.4 Discussion 63
4.6 Related Work 64
4.7 Conclusion 66

II Execution 67

5 Efficient SCOOP 69
5.1 The SCOOP execution model 70
5.1.1 Reasoning guarantees 70
5.1.2 A naïve implementation 70
5.1.3 Issues with blocking 71
5.2 A model with less locking 71
5.2.1 A family of models 74
5.2.2 Deadlock behaviour in different family members 75
5.2.3 Multiple handler reservations 77
5.3 Compiling SCOOP/Qs programs 79
5.3.1 SCOOP/Qs calling convention 79
5.3.2 Request processing 79
5.3.3 Client requests 80
5.3.4 Wait conditions 82
5.3.5 Multi-reservation separate blocks 83
5.3.6 Removing redundant synchronization 84
5.4 Qs run-time design 87
5.4.1 Task layer 87
5.4.2 Schedulable task layer 89
5.4.3 Handler layer 91
5.4.4 EVE run-time comparison 92
5.5 Related work 95
5.6 Conclusion 97

6 Evaluation and comparison **99**
6.1 Benchmarking 99
6.1.1 Parallel 100
6.1.2 Concurrent 100
6.1.3 Setup 101
6.2 Optimization comparison 101
6.2.1 Parallel 101
6.2.2 Concurrent 105
6.2.3 Summary 106
6.3 Implementation comparison 107
6.3.1 Parallel 107
6.3.2 Concurrent 110
6.3.3 Summary 111
6.4 Language comparison 111
6.4.1 Parallel benchmarks 113
6.4.2 Concurrent benchmarks 115
6.4.3 Summary 117
6.5 Related work 117
6.6 Conclusion 119

7 Conclusion **121**
7.1 Deadlock 121
7.2 Concurrent Testing 121
7.3 Execution 122
7.4 Evaluation 122
7.5 Summary 122
7.6 Future work 123

Abstract

There is often a drive to produce efficient programs; efficient programs finish sooner, and can accommodate more advanced features with the extra processing time. Modern hardware makes the use of concurrency to improve efficiency a natural decision. However, to construct a correct concurrent program, one must constantly be aware of the entire system: which threads are modifying which variables, which resources are associated with which locks, and which order locks are taken in. And all of it is non-deterministic. The tension between correctness and efficiency makes concurrent programming a challenging endeavour.

This dissertation addresses correctness issues that are found in concurrent programs and ensures that the execution is efficient so that it is competitive with other, less safe, approaches. For this work, the chosen base language is SCOOP, an extension of the Eiffel programming language. SCOOP is an existing approach that uses the type system to ensure that data races are eliminated. Eiffel pioneered the inclusion of pre- and postconditions as executable parts of the program. The work focuses on three major aspects: the elimination of deadlocks via type checking, the discovery of atomicity violations by repeatable and modular testing, and lastly the enhancement of the SCOOP semantics in a way that allows improved performance through applying specific optimizations at compile and run time.

The existing guarantee of data race freedom is extended through the type system to also include deadlock freedom. This is done by adding annotations to routines that specify what will be locked in the body of the routine, as well as a specification of local order relation that the routine assumes. If the locks are taken as they are specified in the order relation then the Coffman condition of no “cyclical waiting” is satisfied and no deadlock can occur. The type system is formalized in the Coq proof assistant and the core property of deadlock freedom is proven. Additionally, an implementation of the type checker is provided along with inference rules to alleviate the annotation burden, and shown to work on a SCOOP web server application.

Although data race freedom is provided by SCOOP, it is still vulnerable to high-level data races and atomicity violations. To address this, a new technique, demonic testing, enables portions of concurrent systems to be tested deterministically, and in isolation, given that appropriate annotations are provided that describe the interference from the environment. The dynamic state from the program is fed to a new reasoning tool, demonL. The demonL tool and language use an SMT solver as a back end to efficiently find sequences of actions from other threads that would cause a fault in the routine under test. The demonic testing technique is then validated by drawing on a number of examples from a collection of representative concurrency bugs from well known applications (Apache, Firefox, etc.).

The final component of the work is a performance analysis of the semantics of the base language, SCOOP. Several refinements of the SCOOP semantics are proposed that enable greater performance. The semantic refinements are complemented by static analysis to discover and eliminate unnecessary operations and increase performance further, as well as low level run-time optimizations that contribute greatly to the efficiency of the execution. All of the above techniques are combined in a compiler and run-time called SCOOP/Qs which is a fresh implementation of the SCOOP model, and also many ideas are incorporated in the research branch of the EiffelStudio IDE, EVE. The techniques are evaluated on 11 benchmark programs, and compared against 4 other languages (C++, Go, Haskell, and Erlang) to validate the efficacy of the approach. The improvement of SCOOP/Qs over the existing EiffelStudio implementation of SCOOP averages two orders of magnitude.

These techniques target the major problems of concurrent programming, deadlock and atomicity violations, combined with providing data race freedom in an efficient manner. It is only by analyzing and addressing correctness issues *and* designing execution techniques that an effective concurrent programming approach can be constructed.

Zusammenfassung

Effiziente Programme zu erstellen ist oft von grossem Interesse; effiziente Programme werden schneller ausgeführt und die gewonnene Zeit kann für eine Erweiterung ihrer Funktionalität aufgewendet werden. Moderne Hardware macht die Verwendung von Nebenläufigkeit zur Effizienzsteigerung selbstverständlich. Um ein korrektes nebenläufiges Programm zu konstruieren, muss man sich jedoch ständig des gesamten Systems bewusst sein: welche Threads modifizieren welche Variablen, welche Ressourcen sind mit welchen Locks assoziiert, und in welcher Reihenfolge müssen Locks angefordert werden. Und all dies ist nichtdeterministisch. Das Spannungsverhältnis zwischen Korrektheit und Effizienz macht nebenläufige Programmierung zu einer anspruchsvollen Aufgabe.

Diese Dissertation befasst sich mit Korrektheitsproblemen in nebenläufigen Programmen und stellt eine effiziente Ausführung sicher, um mit anderen, weniger sicheren Ansätzen konkurrenzfähig zu sein. Die in dieser Arbeit gewählte Ausgangssprache ist SCOOP, eine Erweiterung der Programmiersprache Eiffel. SCOOP ist ein bestehender Ansatz, der das Typsystem verwendet, um die Eliminierung von Data Races sicherzustellen. Eiffel ist ein Vorreiter der Verwendung von Vor- und Nachbedingungen als ausführbare Programmteile. Die Arbeit konzentriert sich auf drei Hauptaspekte: die Eliminierung von Deadlocks durch Typüberprüfung, die Erkennung von Atomicity Violations durch reproduzierbares und modulares Testen, und schliesslich die Verbesserung der Semantik von SCOOP, um eine Effizienzsteigerung durch die Anwendung spezifischer Optimierungen zur Übersetzungs- und Laufzeit zu erreichen.

Die bestehende Garantie der Data-Race-Freiheit wird durch das Typsystem um Abwesenheit von Deadlock-Freiheit erweitert. Dies wird durch Annotationen von Routinen erreicht, die spezifizieren, welche Locks im Rumpf der Routine angefordert werden, als auch von welcher lokalen Ordnungsrelation die Routine ausgeht. Falls die Locks in der von der Ordnungsrelation spezifizierten Reihenfolge angefordert werden, dann wird die zyklische Wartebedingung von Coffman gebrochen und Deadlocks können nicht auftreten. Das Typsystem wird mit Hilfe des Theorembeweisers Coq formalisiert und die zentrale Eigenschaft der Deadlock-Freiheit bewiesen. Eine Implementierung des Typüberprüfers sowie Inferenz-Regeln zur Reduzierung der Menge der Annotationen werden bereitgestellt und am Beispiel eines SCOOP-Webservers demonstriert.

Obwohl Data-Race-Freiheit durch SCOOP sichergestellt ist, bleibt die Gefährdung durch High-Level Data Races und Atomicity Violations bestehen. Um dem zu begegnen, macht es ein neues Verfahren (Demonic Testing) möglich, Teile eines nebenläufigen Systems deterministisch und in Isolation zu testen, sofern entsprechende Annotationen bereitgestellt werden, die Interferenzen der Umgebung beschreiben. Der dynamische Zustand des Programms wird einem neuen Werkzeug (demonL) übergeben. Das Werkzeug und die assoziierte Sprache verwenden einen SMT-Solver als Backend, um effizient diejenigen Befehlsabfolgen anderer Threads zu finden, die einen Fehler in der zu prüfenden Routine hervorrufen würden. Demonic Testing wird mithilfe von Beispielen validiert, die einer Sammlung von Nebenläufigkeitsfehlern in bekannten Anwendungen (Apache, Firefox etc.) entnommen sind.

Der letzte Teil der Arbeit ist eine Effizienzanalyse der Semantik der Ausgangssprache SCOOP. Mehrere Verbesserungen der Semantik von SCOOP werden vorgeschlagen, um grössere Effizienz zu ermöglichen. Die semantischen Verbesserungen werden durch eine statische Analyse ergänzt, um unnötige Operationen zu entdecken und zu eliminieren und die Leistungsfähigkeit weiter zu verbessern; zusätzlich tragen Laufzeit-Optimierungen stark zur Effizienz der Ausführung bei. Alle genannten Verfahren werden in einem Übersetzer und einer Laufzeitumgebung namens SCOOP/Qs vereint, die eine Neuimplementierung des SCOOP-Modells darstellen; weiterhin werden viele Ideen in EVE, der Forschungsversion der EiffelStudio IDE, integriert. Die Verfahren werden mithilfe von 11 Benchmark-Programmen evaluiert und mit 4 weiteren Sprachen (C++, Go, Haskell, und Erlang) verglichen, um die Effektivität des Ansatzes zu validieren. Die Leistungsverbesserung von SCOOP/Qs gegenüber der bestehenden EiffelStudio-Implementierung von SCOOP beträgt im Durchschnitt zwei Grössenordnungen.

Diese Verfahren behandeln die Hauptprobleme der nebenläufigen Programmierung, Deadlocks und Atomicity Violations, zusammen mit einer effizienten Sicherstellung der Data-Race-Freiheit. Ein erfolgreicher Ansatz für nebenläufige Programmierung muss sowohl auf der Analyse und Behandlung von Korrektheitsproblemen als auch auf dem Design von Ausführungsverfahren beruhen.

Chapter 1

Introduction

Writing concurrent programs is not a new problem, the field has been studied for decades. However, the environment in which concurrent programs are being written *has* changed drastically in recent years. It was never the case before that the average person would have more than one processor in their computer. Now people carry more than one processor in their pocket.

The ubiquity of concurrent programming drives the need for better techniques for managing the complexity of the development process. The measure of success for any software project is simply: does this software fulfill it's functional and non-functional requirements. To fulfill the functional requirements, a program must produce the correct outputs.

In a concurrent program this means the program must be free of data races and atomicity violations. However, the execution of preemptive concurrent programs is non-deterministic. Therefore, determining that a concurrent program produces the correct output is very difficult since the output is at the mercy of the scheduler; one execution may be correct, and the next may not be. This also means that critical errors may lie dormant in software that was previously thought to be well tested, but just hadn't hit a schedule that revealed the bug yet.

Additionally, the program must make progress. Again in a concurrent context, this means it must not deadlock. The same problem of non-determinism affects deadlock, and the execution order of a program is important to determine if a program deadlocks or not. Deadlocks typically occur in multithreaded systems when different threads acquire critical sections in conflicting orders. This is a difficult to detect problem because the root cause is typically not evident from the thread that has deadlocked, but one must rather examine the *other* threads in the system to find which has violated the order of critical section acquisition.

A typical non-functional requirement for a concurrent program is to be efficient, or rather a non-functional requirement of efficiency often necessitates a concurrent program. One critical performance problem in concurrent programs is that programs that share data must do so carefully, for example using critical sections. If the section is highly contended, this leads to a double performance penalty: not only does the thread have to wait to enter the critical section, it must switch to a waiting mode, an operation incurs a context switch requiring additional processor resources to suspend and reactivate another thread.

Therefore, there is a particular kind of tension between the necessity of concurrent programming to achieve greater speed, and the difficulties that constructing those programs encounter. Any technique that aims to ensure correctness cannot be considered independently from execution efficiency; and one cannot gain efficiency by sacrificing critical execution guarantees such that achieving correctness is made too difficult. Balancing these considerations is at the heart of research in concurrent programming.

1.1 Approach

This dissertation's main goal is to develop a comprehensive solution for making concurrent programming more approachable, by addressing both correctness and execution efficiency concerns. The major correctness issues are the existence of deadlock as a result of resource acquisition, data races when two threads modify a shared resource without protection, and atomicity violations as a result of insufficient control of a resource. The execution model for an approach may directly prohibit some classes of errors, so the major impediments to efficient execution have to be discovered by semantic analysis and benchmarking. From this, the goal is further refined into three sub-goals

- Extend the type system by providing rules that will ensure a concurrent system remains free of deadlocks.
- Test concurrent programs for data races and atomicity violations through a modular and reproducible mechanism using executable specifications (contracts).
- Ensure the reasoning and safety guarantees that are provided by the execution model while maintaining acceptable performance.

These claims each provide a fundamental component of a complete approach to raising the quality of concurrent programs.

Since the objective of this work is to show a comprehensive approach to concurrent programming, by considering both correctness and execution, it is natural to have a common programming language upon which to base the contributions. This allows the results to be consolidated and compared against a single frame of reference, making those results more understandable and cohesive. Additionally, the language should represent a paradigm which is well known and often used, and would ideally have good support for executable specifications. For this purpose, the SCOOP language[1] is used, an extension to the object-oriented Eiffel programming language. Eiffel was the first to include contracts as part of the language.

There is a pattern in programming language design to take common errors and make them either difficult or impossible via technical improvements and restrictions imposed by the programming language. This can be seen in the use of structured programming techniques to make control flow more understandable [28, 13], and automatic memory management [63] so that resources do not have to be explicitly managed. Such language-based safety aides are not widely used for concurrent programming. However, SCOOP is a language that eliminates the existence of data races, and thus is a suitable base upon which to extend and build new techniques.

Building on SCOOP, which already uses the type system to distinguish *far references* [69], it is natural to extend the type system to provide even more execution guarantees. One way to prevent deadlock is to prohibit cycles in the locking graph [22]. Since this is a rather straightforward and uniform property

[1]SCOOP is actually a programming model based on Eiffel, rather than a language unto itself. The term language is used here because it does not cause confusion and allows comparison with other languages.

for a program to have, it is a candidate to be a property that the type system may ensure, as typically type systems ensure simple consistency properties.

The elimination of data races, a very important component of constructing correct concurrent programs, does not guarantee that a program will not encounter logical consistency errors due to non-deterministic thread interleavings. These atomicity violations are a more complicated property of a program, and are defined in terms of the invariants that are specific to a particular program. For this reason, testing for atomicity violations is a suitable approach, as testing uses the definitions in the program to determine correctness, and tests are necessarily defined using the functions and routines provided by a specific program. Using a language which supports contracts is also essential here as it provides a way to precisely communicate the notion of what an error is.

Providing techniques to ensure the correctness must be balanced with the cost of such techniques at run-time, since a very common goal of using concurrent execution is to increase performance. Since the balancing act is between the correctness guarantees and the extra work necessary to attain them, the approach to ensuring efficient execution must include an analysis of how the guarantees are ensured by the semantics of the language. However, this is only the first half of the job, the second half is to implement the semantics efficiently and to trim any unnecessary overhead down as much as possible.

1.2 Contributions

The above goals are achieved by producing several contributions to the state of the art. In particular, in the areas of deadlock freedom, concurrent testing, and the efficient execution of a concurrent object-oriented language while maintaining data race freedom.

Deadlock freedom. Deadlock freedom is achieved statically by a modification of the SCOOP type system. The modified type system tracks resource usage and ensures that every time a new handler is requested to apply a method it does so in a way that is consistent with all the other handlers in the system. Consistent in this case means corresponding to the same partial order. The locking behaviour of SCOOP programs is formalized in an operational semantics and the restrictions of well-typed programs is proven to ensure deadlock freedom during the execution according to the semantics. A tool checks the validity of programs in the type system and also infers program annotations to reduce the burden placed on the programmer. The core aspects of the proof of soundness are formalized and proven in the Coq proof assistant.

Concurrent testing. demonL is a testing technique which assumes the worst will happen in a given concurrent program. It uses the information present in a contracted program to construct a domain of actions that can occur at any point in time. This domain of actions is then utilized at run-time to take the state and discover sequences of actions from the domain that could cause a precondition violation. This draws on the ideas of rely-guarantee reasoning. A symbolic reasoning tool, demonL, is constructed and uses an SMT solver to reason about the interference that may occur given a particular initial state and a particular domain of actions. This allows both data races and atomicity violations to be detected, and since demonL is deterministic, the tests are repeatable and can even be driven by existing unit tests. The technique is evaluated on different classes of concurrency bugs from well known projects, with good detection rates and execution times to demonL to synthesize interference.

Efficient execution of SCOOP. The data race guarantees that SCOOP provides and existing semantics are first analyzed to discover new optimization opportunities. Based on this, a modified SCOOP semantics is developed that can achieve greater performance. A fresh SCOOP compiler, run-time, and specific optimizations, together referred to as SCOOP/Qs, are implemented to test the efficacy of the semantic changes, low-level run-time techniques, and optimization opportunities. The same techniques are integrated, where possible, in the research branch, EVE, of the EiffelStudio run-time. Executing the modified SCOOP model on a variety of a wide variety of benchmark programs verifies the effectiveness of the techniques. When the techniques are implemented in EVE, the performance increases by an order of magnitude over EiffelStudio. The customized SCOOP/Qs implementation attains a speedup of two orders of magnitude over EiffelStudio. Other popular concurrent programming languages are also included in the comparison to give a better and more accurate picture of what performance one should expect from different approaches to concurrency with different safety guarantees. SCOOP/Qs is found to be the overall fastest of the languages that provide data race freedom.

1.3 Organization

The common background for all of the work is given in Chapter 2. This includes the concepts of object-oriented programming, program specification and contracts, the Eiffel language, how threading systems work, and also an explanation of the SCOOP model. Chapter 2 also defines the different types of concurrency bugs.

The approach to preventing deadlock in SCOOP programs is given in Chapter 3. This includes a formalization of the SCOOP semantics focusing on the locking behaviour, a proof sketch of correctness, and also a formal proof of the core aspects of the technique.

Demonic testing, an approach which uses advanced reasoning tools to construct interference, is described in Chapter 4. The chapter additionally provides an explanation of the demonL tool and language, as well as how it is translated into a satisfaction for the Yices SMT solver to process.

Chapter 5 gives a modified SCOOP semantics, SCOOP/Qs. The relation between the semantic and compilation and run-time is explored, along with justifications for the different design decisions made for each of these aspects of the SCOOP/Qs approach.

For validation of the technique, Chapter 6 provides extensive benchmarks comparison optimizations, language variants, and compares entirely different languages with regards to performance on concurrent and parallel tasks.

Chapter 7 draws conclusions of the work.

Chapter

2 Background

This chapter outlines the key concepts that underlie this work. This includes a definition of object-oriented programming, program specifications and contracts, the threaded concurrency model, the Eiffel and SCOOP languages, and different kinds of concurrency bugs. It is important to have a grasp of these concepts to have a more full understanding of the rest of the thesis.

2.1 Object-oriented programming

There are many languages that are considered to be "object-oriented". To avoid confusion, the definition of object-oriented used for this work is taken from Pierce [84]. An object-oriented language allows:

Multiple representations An object determines what code is executed when invoking a method on it (dynamic dispatch). In contrast, an abstract data type (ADT) has only a single implementation.

Encapsulation An object hides its internal representation, manipulation occurs by invoking methods on the object. ADTs are similar in this regard.

Subtyping An object's type is determined by its interface, the list of names and types of its methods. Any superset of an interface is likewise an interface, and any object implementing that super-interface will be a subtype of an object implementing the sub-interface. This is because any operation that can be called on the sub-interface can be called on the super-interface, so objects with the super-interface can substitute for ones with the sub-interface.

Inheritance Inheritance is a mechanism to reuse interfaces and implementations. Often this is done via classes and subclassing is used to reuse a class' implementation. Methods can also be overridden during subclassing, providing for new (refined) behaviours.

Open recursion There is often a special variable called `Current` or `this`, that gives the ability for superclasses to invoke (via dynamic dispatch) subclass implementations of the superclass' interface.

These properties relate to software qualities which are desirable. Subtyping increases code reuse by allowing client code to operate on any object that has a subtype relation with the static types it expects. Dynamic dispatch allows this same client code to produce different results according to each subtype it is passed. Inheritance is another mechanism for code reuse, this time for the supplier instead of the client; class implementations can not only be reused, but also extended using inheritance. Encapsulation allows for information hiding [81], meaning lower coupling between classes or modules.

Often, there is an invariant associated with a class that must always hold. However, it is not always be possible for the class' methods to maintain the class invariant by themselves. For example, the invariant of a hash table requires that the hash codes of the stored keys do not change, else the location of the key will no longer associate properly to the key's hash code. If the keys are mutable objects, then the rest of the system must not manipulate any of the key-objects in a way which causes their hash codes to change.

2.2 Specifications and contracts

Software specifications are a way of defining how software should behave. A common method of specifying software is to affix preconditions, postconditions, and other invariants to the system. Preconditions describe what must hold immediately before a group of statements. Postconditions describe what the routine must ensure as a resulting state after it has completed.

These specifications are combined with Hoare-style axiomatic semantics [42] to reason about program correctness. Verification tools such as VCC, Jessie, Boogie, and Verifast (using separation logic [86]) use this method to prove that a program's implementation corresponds to its specification. There are languages, such as JML [56] and ACSL [6], that formally describe how to write these specifications. The basic scheme to denote the specification for a program element s is $\{P\}\ s\ \{Q\}$, where P is the precondition and Q is the postcondition. For example, the schema for assignment in Hoare logic is

$$\{P[E/x]\}\ x := E\ \{P\}$$

expressing that if x in the expression P can be replaced by E in the prestate, then P holds in the poststate of the statement $x := E$. Given the instantiation of the schema

$$\{x + 1 > 4\}\ x := x + 1\ \{x > 4\}$$

it must be that $x > 3$ in the prestate to satisfy condition that $x > 4$ in the poststate.

Design by Contract [66] refers to the usage of executable preconditions, postconditions, and class invariants. The invariants are compiled and executed at run-time, triggering an error if they violated. The technique is used to help ensure software quality by both stating the requirements in the program text, and also providing constant feedback during execution as to whether the invariants hold.

Although first appearing in Eiffel, this technique has seen usage in other languages as well, such as in D [1] with built-in support. Other languages, such as C# with Code Contracts [21], C++, Groovy, Java, Perl, Python, and others, offer contracts through a library. In the end, the objective is to put the code's specification the code, and have it checked at run-time to ensure that the two correspond.

2.3 Eiffel

Eiffel [65] is an object-oriented programming language and was the first language to introduce contracts. It offers multiple inheritance and generics as modelling and software reuse mechanisms.

The syntax typically easy to understand without much explanation, using full words as keywords. Types are denoted using `x: T` to declare that the entity `x` is of type `T`.

Routine contracts in Eiffel are specifications in the form of pre- and postconditions as part of `require` and `ensure` clauses, respectively. The `old` keyword indicates that the value following it will be considered from the pre-state of the routine execution.

2.4 Threaded model

Threading is by far the most dominant model of concurrency. It is the default model in C, C#, C++, Java, Objective-C, Python, languages which cover much of the software written today. Each thread is identified by an independent call stack, set of registers, and thread state. This enables each thread to make calls, use local storage from the stack, and change its thread state (running, waiting, etc) without interference from other threads. A program is identified by a shared heap, and at least one associated thread. The global state, the process' shared heap and any state existing outside the process, is shared between threads. Typically it is the modification of global state that makes threaded programming difficult: careless modifications to the global state by one thread can violate the assumptions another thread has made about the same state. Threads can be implemented in different ways.

The Linux kernel's approach to threading is used here to demonstrate how threads may be implemented in an operating system. Linux Kernel Development [59] states that Linux threads and processes share an implementation. Both processes and threads are created by calling the `clone` system call, allocating and initializing the `task_struct` structure to represent the new thread of execution. The `task_struct` contains most of the important task information, such as:

flags different properties that the thread has, such as being running with super privileges or not being called from `exec`.

mm virtual memory information. This can be used to separate memory spaces such as the separation seen between separate processes, or to share memory spaces as is the case with threads running in the same process.

files file descriptor information. Much like the virtual memory information, this can either be used to share or distinguish the ownership of file handles for each task.

status the current status of the thread, the standard options being one of `TASK_RUNNING`, `TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE`, `TASK_ZOMBIE`, or `TASK_STOPPED`

With this representation, the differences between thread or process creation are just the options given to `clone`. One set of options will allocate new file

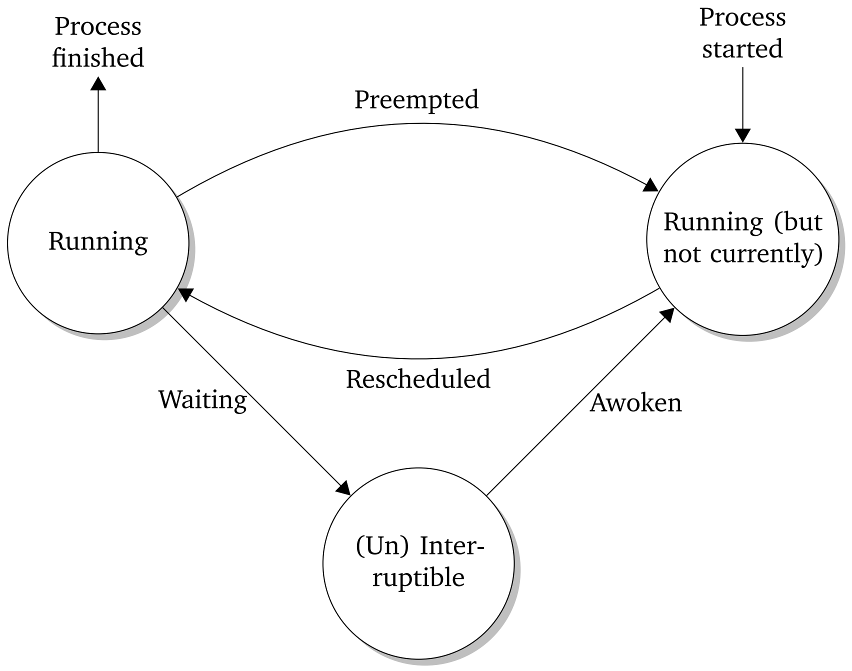


Figure 2.1: Linux task state diagram.

descriptor and memory mapping information to create a process, the other will copy this information from the creating process and thus share file and memory spaces.

The list of all tasks is held in the kernel, and the kernel is responsible for switching between the tasks according to some policy. There are different scheduling policies, the current Linux default is the CFS (Completely Fair Scheduler). The basic responsibility for a scheduler is to decide when one task has had enough time on the CPU and to choose the next task to execute.

Figure 2.1 shows the task state-transition diagram for the Linux kernel. When created, a task is added to the list of tasks. Initially, it is runnable but not yet running. The `state` field does not distinguish if a task is currently running or not, both are marked as `TASK_RUNNING`. The distinction is made by whether the task is in the list of runnable tasks or not. The task can also be moved to the `TASK_INTERRUPTIBLE` state, for example if it tries and fails to acquire a mutex. If the mutex is then released, the task may be awoken and moved to the `TASK_RUNNING` state and added to the runnable tasks list. A non-waiting task may also be moved to the runnable list if the scheduler has decided it has run long enough and another task should be given some CPU time. Likewise, in the future it will be rescheduled and removed from the runnable list.

A task is in the `TASK_STOPPED` state if it has been sent a signal to stop. Debuggers use this signal to stop currently running applications so the developer can examine their state; it can also be triggered explicitly using a program like `kill`. The `TASK_ZOMBIE` is used to indicate a task that stopped but not yet collected by the task manager. Tasks will remain in this state until their parent either receives the notification that the task has stopped, or notified the kernel that it does not want to know about such information. After this is known, all data related to the task will be freed by the kernel. TASK_STOPPED and TASK_ZOMBIE, do not appear in Figure 2.1, as they are not part of the basic life cycle of a task, but they do complete the explanation of the Linux task management system.

2.5 SCOOP model

SCOOP (Simple Concurrent Object Oriented Programming) is an extension to the Eiffel language. First appearing in [67], and later refined in [78] and [70], SCOOP's addition to Eiffel consists of a single keyword: `separate`. The `separate` keyword plays dual roles as both a type modifier and a new type of block instruction (like `if` or `check` instructions). As a modified type, the type `separate C` denotes the set of objects that *may* execute their methods concurrently with other objects. To exclude the possibility of data races, a client may only interact with a `separate` object when that `separate` object is *controlled*. `separate` objects are only controlled when they appear as arguments to a routine or to a `separate` block, as in Figure 2.2. In previous work [67, 78], `separate` objects could only be controlled by appearing as arguments to a routine, this work introduces the `separate` block as a convenience to save writing new routines every time objects need to be controlled.

Example 1 It is best to first look at an example to build an intuition for how SCOOP programs work. In Figure 2.2 there are two SCOOP programs running in parallel. Supposing that x is the same object in each thread, there are only two

```
separate x
  do
    x.foo()
    a := long_comp()
    x.bar()
  end
```

Thread 1

```
separate x
  do
    x.bar()
    b := short_comp()
    c := x.baz()
  end
```

Thread 2

Figure 2.2: A simple SCOOP program

possible interleavings of operations on x:

- `x.foo(), x.bar(), x.bar(), x.baz()` or
- `x.bar(), x.baz(), x.foo(), x.bar()`

However, in contrast to `synchronized` blocks in Java, `separate` blocks control access to shared memory by controlling the concurrent actions that can act on that shared memory. For example, in both threads, the calls on x are performed asynchronously, thus for Thread 1, `x.foo()` can execute in parallel with `long_comp()`. However, it *cannot* be executed in parallel with `x.bar()` as they have the same target, x. SCOOP has another basic operation, the query, that provides synchronous calls. It is so called because the sender expects an answer from the other thread; this is the case with the `c := x.baz()` operation, where Thread 2 waits for `x.baz()` to complete before storing the result in c. Using the traditional SCOOP rules the `separate` blocks in Figure 2.2 would have to be separate functions:

```
r1 (x: separate X)
  do
    x.foo()
    ...
  end
```

The SCOOP model associates every object with a thread of execution, called its *handler*.[1] There can be many objects associated to a single handler, but every object has exactly one handler. In Figure 2.2, `x` has a handler that takes requests from Threads 1 and 2. Some treatments of SCOOP, such as [78], define a type expression `x.handler` to denote the handler of object `x`, although this has not been included in any current implementation of the SCOOP model.

A client must register its desire to log requests on any handler that it will subsequently make calls on. This is normally done when a routine has `separate` arguments; the client will register with all handlers of the separate arguments when the routine is called. A second syntax is also used in this work, namely the `separate` block syntax. The semantics of a `separate` block are the same as the SCOOP-specific parts of the semantics of a routine call. The client threads are deregistered at the end of the `separate` block in Figure 2.2, just as it happens at the end of a routine with separate arguments.

There are both asynchronous and synchronous calls that can be logged on a handler. An asynchronous call is one where the routine that is logged has no result. Since there is no result, the client cannot inspect the effect of the call except by logging a query afterwards. Queries (methods that return a result) are synchronous. Since the client may want to use the value returned by the query in a computation it must wait for the handler to compute this value before continuing.

Queries with `separate` targets can also be used in the precondition of routines (also in the `separate` blocks). When `separate` queries occur in a precondition they are transformed into wait conditions. A wait condition will not proceed into the body of the routine (`separate` block) until it evaluates to true. When control passes into the body of the routine, the body can assume the wait condition is true and the target is reserved (i.e., no other calls could make it false in the meantime).

Example 2 Given a `separate BUFFER[INTEGER]`, a routine to extract an element from the buffer must consider that the buffer may be empty. In such a situation, it would use a wait condition as in

```
extract(b: separate BUFFER[INTEGER])
  require
    not b.is_empty
  do
    ...
  end
```

[1]The term *processor* was used in prior literature on SCOOP to refer to this concept. *handler* was also used previously, and replaces the uses of *processor* in this work to avoid the ambiguity of the term processor.

to wait until the buffer had an element. SCOOP guarantees that the body of the routine can then extract at least one element.

The SCOOP type system is a *relative* type system. If a class `C` has a query, `f` with the static result type `D`, and `f` is called on a target of type `separate C` then this will not return `D`, but rather `separate D`. Since the return type is non-`separate`, this means it resides on the same handler as the target. However, if the target is `separate` relative to the caller, then the result of the query must also be `separate` relative to the caller, because it is "with" the target. This transformation prevents `separate` objects from being stored in non-`separate` references. The relation between the target type and return types are given in Table 2.1.

		Result	
		non-`separate`	`separate`
Target	non-`separate`	non-`separate`	`separate`
	`separate`	`separate`	`separate`

Table 2.1: Result type transformation

When data flows from clients to handlers, as arguments to `separate` calls, a similar transformation has to be made. Again, this is to preserve the soundness of the type system. The actual arguments of the call are transformed as shown in Table 2.2, the transformation is done based on the formal argument and target types. Bottom ($\perp$) appears in the case where the target is `separate` but the formal argument is non-`separate`. The type system cannot distinguish between handlers, so there is no way to guarantee that the actual argument given would belong to the same handler as the target. Because of this, no type can satisfy it; in essence this is a typing error. Expanded types (`INTEGER`, `REAL`, etc) are immune to type translation as their semantics is to be copied entirely when sent (received) to (from) a handler.

		Formal argument	
		non-`separate`	`separate`
Target	non-`separate`	non-`separate`	`separate`
	`separate`	$\perp$	`separate`

Table 2.2: Actual argument type transformation

The SCOOP model is similar to message passing models, such as the Actor model [40]. What distinguishes SCOOP from languages like Erlang [4] is that the threads have more control over the order in which the receiver will process the messages. When multiple processes each send multiple messages to a single receiver in Erlang, the sending processes do not know the order of processing of their messages (as they may be interleaved with messages from other processes). In SCOOP, since each thread registers with the receiver, the messages from a single separate block to its handler will be processed in order, without any interleaving.

This ordering gives the programmer the ability to reason about concurrent programs in a sequential way within the `separate` blocks. To be precise, pre/-postcondition reasoning can be applied to the `separate` objects protected by the `separate` block, even though the actions are being executed in parallel. `separate` objects are marked as such by the type system, and one may only call methods on a `separate` object if it is protected by a `separate` block. Since the calls that the client wants to log are only applied by the handler, this rules out the possibility of low-level data races. However, high-level data races and liveness are not guaranteed by the SCOOP model.

Formalizations of the SCOOP model take different approaches, Ostroff et al. [79] define a SCOOP virtual machine (SVM), Brooke et al. [15] use Communicating Sequential Processes (CSP), and Morandi et al. [73] define a comprehensive operational semantics and use the Maude term-rewriting system to explore the behaviour of the model.

2.6 Types of concurrency bugs

Concurrent programs introduce a unique set of bugs that cannot arise in a sequential setting. In *Foundations of Multithreaded, Parallel, and Distributed Programming* [2] safety and liveness, the properties that state that nothing bad ever happens (safety), and something good eventually happens (liveness), are compared to functional correctness and termination, respectively.

Data races. If at least two threads access the same memory location, and at least one thread issues a write to that location, and the operations are not ordered by some synchronization mechanism (such as a mutex), then this is a data race. Data races are one of the most basic types of concurrency errors in the sense that they represent the default way of accessing memory in a shared memory multithreaded program.

Deadlock. The typical way to achieve exclusive access to a resource is to use a mutex or other synchronization primitive. By design, these primitives block the progress of the other threads while a given thread has exclusive access to the resource. If there is a group of threads that are each waiting to acquire a resource that another member in the group already has, then there is a cyclical dependency among the threads in the group. The result of this is that none of the threads can make progress; this is called deadlock. Deadlock prevents a basic liveness property, that processes always terminate, from being satisfied.

Atomicity violations. Errors are still possible even when synchronization is used to provide freedom from data races. Synchronization is often used to provide mutual exclusion to some shared resources for a finite amount of time. However, sometimes the mutual exclusion does not last long enough to guarantee correct operation, i.e., a mutex is released, after which time it is re-acquired with the assumption that nothing has changed since the previous release. When this happens it is called an atomicity violation.

Order violations. There are programs that have several threads where it is assumed that one thread always runs certain operations before another thread. For example, it could be that a worker thread must complete a job before the main thread uses the result. An order violation occurs when dependent operations occur out of sequence.

Studies show [60] that most (97%) non-deadlock, non-data race bugs are either atomicity or order violations. The absence of data races, deadlock, atomicity and order violations are all safety properties of a concurrent system.

Part I: Correctness

Chapter
3 Static deadlock detection

The type system is the first line of defence in program correctness. The typical static type system allows properties about the program to be stated and checked at compile time. Generally these properties are relatively simple, although they can also be much more elaborate as in the use of dependently typed languages such as Agda, Epigram, Idris, and Coq, however such systems take much more effort to produce programs that type-check. There is a balance to be struck between the expressive power of the type system and the effort required to satisfy the type checker. This chapter shows that the absence of deadlocks, since it can be stated as only a moderately complex property, can be ensured statically through an extension to SCOOP's type system.

The problem of deadlock in concurrent systems can also be handled, dynamically, at run-time. Research into concurrent programming has provided a set of tools, e.g. [53, 90, 27, 14], for addressing the data races and deadlocks that arise from incorrect use of synchronization. However, dynamic approaches represent a reactive approach to finding deadlocks, where this work advocates a proactive, preventative, approach.

Of course the situations in which deadlock can arise are connected to the execution model of the underlying programming language. For SCOOP, the mechanism that guarantees data race freedom, handler reservation, is the same mechanism that can cause deadlocks, because the existing operational semantics [78] mandates the use of locks to achieve exclusive access to the handlers. In Figure 3.1, through the call to `d1`, `a` is locked by the call to `g (a)` in the body of `f`, and `b` is locked by the call to `h (b)` in the body of the routine `g`, so in the (empty) body of `h`, locks on the handlers of both `a` and `b` are held. The program in Figure 3.1 may deadlock; `d1` and `d2` lock `a` and `b` from `MAIN` in the opposite order.

On one hand, the handler reservation mechanism simplifies reasoning about concurrent programs because it eliminates data races. On the other hand, SCOOP offers little protection against deadlocks, a property shared with many concurrent programming languages, that may arise due to misuse of the handler reservation mechanism.

The proposed approach extends the SCOOP type system by enriching the routine interface specifications to establish an intended order in which handlers can be reserved. Although the type system is extended, it retains its modularity, thus the resulting deadlock prevention technique is also modular. It is important that these extensions to the type system actually guarantee that the program is free of deadlock; the technique is proven to work by first formalizing the SCOOP locking behaviour in a structural operations semantics. A proof of soundness then uses the operational semantics to show that at every program step deadlock freedom is preserved. The core of this proof is formalized in the Coq proof assistant to ensure the validity of the soundness reasoning. The technique

```
class DEADLOCK
create set
feature
  x, y : separate S

  f
    do
      g (x)
    end
  g (a : separate S)
    do
      h (y)
    end
  h (b : separate S)
    do
    end
  set (a_x, a_y : separate S)
    do
      x := a_x
      y := a_y
    end
end
```

```
class MAIN
feature
  a, b : separate S

  run(d1,d2: separate DEADLOCK)
    do
      d1.f
      d2.f
    end

  make
    local
      d1, d2 : separate DEADLOCK
    do
      create a
      create b
      create d1.set (a, b)
      create d2.set (b, a)

      run (d1, d2)
    end
end
```

Figure 3.1: SCOOP deadlock example

has been implemented in a type checker and applied to a simple web server programmed in SCOOP.

Other work in this area such as deadlock freedom for active objects in Java [50] provides less versatile structures (trees vs. orders). Techniques of similar power [14], however, are not applied to an underlying language that guarantees other correctness guarantees such as data race freedom. Lastly, other partial operational semantics [79] only consider liveness properties in the light of model checking, which may not scale to larger programs.

Section 3.1 gives an overview of how deadlocks arise in SCOOP and how they can be detected both dynamically and statically. Section 3.2 provides a formalization of SCOOP's locking semantics. Section 3.3 describes the deadlock prevention scheme and prove that well-formed programs cannot deadlock. Section 3.5 provides a mechanized proof of the core argument that the technique is sound. Section 3.6 compares the related work and Section 3.7 contains a discussion of the approach.

3.1 Deadlocks in SCOOP

When a SCOOP program deadlocks, it is tied to the order in which handler reservations are made between different calls or different `separate` blocks. It is instructive to look at the key aspects informally first to understand the mechanisms that work to create deadlocks, then to see a case where a real deadlock may arise.

3.1.1 Types and handlers

A variable x is declared of separate type

```
x : separate X
```

then at creation of x with the statement

```
create x
```

a new handler p is created in addition to an object o of type X, and the handler of o is set to p. The declaration x: separate X just means that x is an object which may be on any handler. The type system presented in SCOOP'07 [78] allows the types of separate objects to be made more precise by indicating exactly which handler an object is on. Given the following,

```
x : separate <p> X
y : separate <p> Y
```

at run-time when the objects are constructed, the same handler, p, would be responsible for both x and y. The handler annotations have class-scope if applied to attributes of the class, and routine-scope if applied to local variables of a routine. Handler tags are not currently in the SCOOP implementation provided in EiffelStudio. This deadlock prevention approach resurrects this type annotation as it allows handlers to have a representation at the type level, and thus they can appear in other predicates, which will be required.

3.1.2 Locking through calls

As outlined in Section 2.5, a client must first reserve a handler before logging calls on that handler. However, the mechanism that actually guarantees this reservation was only mentioned briefly in this chapter.

The assurance that the calls are processed by the handler without interleaving from other clients has is provided in the existing SCOOP model [78] by associating a lock with every handler, and taking that lock when a reservation is needed. This lock is held for the duration of the call that reserved the handler, and released at the end of the call's body. In Figure 3.1 the body of the feature f contains the command g(x), the locking behaviour described above would be seen here, as this call is invoked, requesting and locking the object x.

This gives the basic locking behaviour of SCOOP programs, and from here it is possible to see where deadlocks can arise.

3.1.3 Example

When encountering handler reservation f (x), where x has a separate type, first the lock on x is requested, and if/when it is available it may be given the client requesting it. A deadlock state, based on waiting for resource availability as in [22], can be identified

- dynamically: construct a "waits-for" relation; if an element is related to itself in the transitive closure of such a relation, then the system is in a deadlock state. In the setting of SCOOP, the "waits-for" relation contains an association between between handlers p and q iff some other handler has a lock on p and is requesting q.

- statically (conservative): arrange the handler tags into a partial order. When the text of the program indicates a lock is taken, verify that it is less than all the other locks that could have been taken at this point. The program text may require some annotations establishing which locks have already been taken. This is essentially a way to ensure that the dynamic case never arises when the program is run.

These two schemes can be applied to Figure 3.1. Reasoning using the dynamic scheme, an instance of class `DEADLOCK` will lock its attributes `x` and `y` in some order when its routine `f` is called. In class `MAIN`, two instances `d1` and `d2` of `DEADLOCK` are initialized with two separate objects `x` and `y`, however their order is reversed between the two instances. By executing `run`, the routine calls `d1.f` and `d2.f` are executed asynchronously, according to the semantics of calls on separate objects `d1` and `d2` outlined above.

As a result of executing `d1.f`, the call `g(x)` is invoked. As `x` is an argument to the routine `g`, the run-time locks `x` for the duration of the call, as prescribed by the semantics for controlled objects outlined above. In particular, `x` will still be locked when the call `h(y)` is invoked, requesting a lock on `y`. The concurrent execution of `d2.f` has an analogous locking behaviour, but since `d1` and `d2` have opposite views of `x` and `y`, the locking order is reversed. Hence the calls may ultimately form a cyclical locking pattern, resulting in a deadlock.

To reason statically about the same sequence of calls, one notices that the order of calls can be conservatively approximated by examining the program text, and observing which routines subsequently call other routines. In the case of Figure 3.1, calling the feature `f` will (for a general routine, may) always require that the handler of `x` is locked, followed by `y`. This information can be used statically at the call sites of `d1.f` and `d2.f` to determine that their concurrent execution could lead to a deadlock state.

3.2 Locking operational semantics

To understand how to construct deadlock-free programs, a clear notion of what a program is must be given. This section presents a partial syntax and semantics that focuses on the locking behaviour of the SCOOP model. Trimming away the parts of the model that do not concern locking allows one to more clearly understand the problem.

3.2.1 Syntax

This section describes the abstract syntax of SCOOP programs. This abstract SCOOP syntax uses the set Tag to denote handler tags, and normally uses $p,\ q,\ r \in Tag$, etc. as example handler tags. Handler tags are the syntactic representations of an actual handler that will be used by the type system. There is also the set $Name$ which are just routine identifiers, often $f \in Name$ is used as an example instance.

The basic component of a SCOOP program is an expression. The expressions syntax is

$$e \ ::= \ [p] \ \big| \ \mathsf{skip} \ \big| \ \mathsf{create}(p) \ \big| \ e \cdot f(\tilde{e}) \ \big| \ e; e \ \big| \ \mathsf{lock}\ \tilde{p}\ e$$

The notation $\tilde{e}$ denotes a sequence of expressions $e_1, \ldots, e_n$, and similarly $\tilde{p}$ denotes a sequence of handlers. Sometimes these sequences are treated as sets where no confusion arises, i.e. $\tilde{e} = \bigcup_{i=1,\ldots,n}\{e_i\}$. The existence of a function that maps expressions to their handlers, $\mathit{tag}_{\mathcal{P}} : \mathit{Expr} \to \mathit{Tag}$, is assumed to be given, and $\mathit{tags}_{\mathcal{P}}$ is the same function except lifted to map over sequences of expressions.

$[p]$ represents a value on a handler. The value itself is not mentioned as the locking behaviour of SCOOP does not depend on particular values, but rather on the handler where those values reside. Sequential expression composition is denoted by $e_1; e_2$. The expression skip has no effect, and is the left identity of sequential composition. A new handler with tag p is created using create(p), again the actual value or object that is created is abstracted away. Calling a routine f on a target t, with a list of arguments $\tilde{a}$ is represented by $t \cdot f(\tilde{a})$. A sequence $\tilde{e} = e_1, \ldots, e_n$ where all expressions are fully evaluated (i.e. $e_i = [p_i]$ for $i = 1, \ldots, n$) is denoted by $[\tilde{e}]$.

The last syntactical element, lock, is not a part of the syntax that can be written explicitly in a program. lock can only arise as a result of intermediate steps in a program's execution. lock $\tilde{p}$ e represents the locks on $\tilde{p}$ taken as a result of a **separate** call; e represents the body of the routine call.

The **separate** block is not included in this syntax because it is nearly identical to a routine call and thus would only serve to go against the goal of this treatment to trim away unnecessary details. Additionally instructions such as **if**, **loop**, **rescue** and **check** are not handled explicitly as they are compositions of the above syntax elements.

Having the expression definition, a *program* $\mathcal{P}$ is a mapping of names to routines

$$\mathcal{P} \in \mathit{Program} = \mathit{Name} \to \mathit{Routine}$$

where a routine rtn is a tuple of the form

$$\mathit{rtn} \in \mathit{Routine} = \mathit{Tag}^* \times \mathit{Expr}$$

and for convenience the projections of this tuple can be named as

$$\mathit{rtn} = (\mathit{rtn}_{\vec{as}}, \mathit{rtn}_{\mathsf{body}})$$

The component $\mathit{rtn}_{\vec{as}}$ is the sequence of formal arguments of the routine. Since the actual values, and by extension their specific types, do not influence locking, only the handler component (Tag) of a type is used. The $\mathit{rtn}_{\mathsf{body}}$ component is the body of the routine, and is an expression.

This treatment of the SCOOP syntax and programs focuses on the locking behaviour of SCOOP programs as given by the routine call. By only allowing the subset of the full SCOOP syntax necessary to lock, and therefore deadlock, the syntax will give rise to a more focused analysis of deadlock behaviour.

3.2.2 Locking semantics

Given the above syntax and structure of SCOOP programs, this section defines the *semantics* of how locks are requested and taken in the SCOOP model. This is forms the basis for the reasoning about how deadlocks arise and how they are prevented.

At run-time, a program $\mathcal{P}$ gives rise to a *process* P which is described by the following syntax:

$$P \ ::= \ p :: e \ \big| \ P \mid P$$

A process is therefore either an expression e located on a handler p, or a parallel composition of processes. The idea is that a program starts with the initial call f_0 on an initial handler p_0 as $p_0 :: f_0$, and will give rise to more parallel threads (as the result of create expressions) as the execution proceeds. A structural equivalence $\equiv$ over processes specifies the commutativity and associativity of the $\mid$ operator; the formal definition of $\equiv$ is standard and not given here. The handler tags are unique within processes, i.e. there cannot be a process $p :: e \mid q :: e' \mid Q$ such that $p = q$. This property is preserved by handler creation, which is the only operation that can introduce a new parallel composition of processes. The expression $p \in P$, where p is a handler and P is a process, is true if and only if the handler p is one of the handlers in the process P.

Processes operate on a state only containing locks and requests. Formally, a *lock state* L is a pair of mappings (L_l, L_r) of the following type:

$$L \in \mathit{LockState} = \big(\mathit{Tag} \to \wp(\mathit{Tag})\big) \times \big(\mathit{Tag} \to \wp(\mathit{Tag})\big)$$

Here, L_l is a mapping from a client handler (tag) to a set of handlers, representing the handlers that a client handler currently locks. Similarly, L_r is also a mapping from a client handler to a set of handlers. However, L_r represents the handlers that a client handler is requesting, not the ones that it has already obtained. Representing the locks and requests this way allows this model to use the same terms as those used in the Coffman deadlock definition [22]. The domain of L is defined as the union of the domains of its components, $\mathit{dom}(L) = \mathit{dom}(L_l) \cup \mathit{dom}(L_r)$. The notation $L_l[p \mapsto T]$ denotes a point-wise update to the map. The update is such that the resulting mapping returns T at point p of its domain and is unchanged otherwise at all other points.

The semantics specifies rewrite rules over processes and lock states in the style of a structural operational semantics with transitions of the form

$$\mathcal{P} \vdash (P, L) \to (P', L')$$

This means that given a program $\mathcal{P}$, providing a mapping from routine names to their bodies, the process P evolves in one step to P' and transforms locking state L to L' also in this one step.

With this background information, the semantic rules relating to locking in SCOOP programs in Table 3.1 can be explained. The creation of a new handler q by client p gives rise to a new parallel handler located at q. If the handler already exists, then this has no effect. These behaviours are expressed in the CREATE$_1$ and CREATE$_2$ rules. For routine target and argument evaluation: EVALTRG and EVALARG enforce that targets are fully evaluated before arguments are evaluated. In EVALARG, the arrow $\twoheadrightarrow$ represents performing a single rewrite step on a sequence of expressions. To reorder the constituent handlers of a program during rewriting, the EQUIV rule is available. The rule SEQ allows one step to be performed on the left side of a sequential composition, and SKIP carries its intuitive meaning.

Once the target and arguments of a call are both fully evaluated, the call can be invoked. The LOGCALL rule is for logging asynchronous calls, and moves the

CREATE$_1$
$$\frac{q \notin (p :: \mathsf{skip} \mid Q)}{\mathcal{P} \vdash (p :: \mathsf{create}(q) \mid Q, L) \rightarrow (p :: \mathsf{skip} \mid q :: \mathsf{skip} \mid Q, L)}$$

CREATE$_2$
$$\frac{q \in (p :: \mathsf{skip} \mid Q)}{\mathcal{P} \vdash (p :: \mathsf{create}(q) \mid Q, L) \rightarrow (p :: \mathsf{skip} \mid Q, L)}$$

EVALTRG
$$\frac{\mathcal{P} \vdash (p :: t \mid Q, L) \rightarrow (p :: t' \mid Q', L')}{\mathcal{P} \vdash (p :: t \cdot f(\tilde{a}) \mid Q, L) \rightarrow (p :: t' \cdot f(\tilde{a}) \mid Q', L')}$$

EVALARG
$$\frac{\mathcal{P} \vdash (p :: \tilde{a} \mid Q, L) \twoheadrightarrow (p :: \tilde{a}' \mid Q', L')}{\mathcal{P} \vdash (p :: [q] \cdot f(\tilde{a}) \mid Q, L) \rightarrow (p :: [q] \cdot f(\tilde{a}') \mid Q', L')}$$

SKIP
$$\frac{}{\mathcal{P} \vdash (p :: \mathsf{skip}; e \mid Q, L) \rightarrow (p :: e \mid Q, L)}$$

SEQSTEP
$$\frac{\mathcal{P} \vdash (p :: e_1 \mid Q, L) \rightarrow (p :: e_1' \mid Q', L')}{\mathcal{P} \vdash (p :: e_1; e_2 \mid Q, L) \rightarrow (p :: e_1'; e_2 \mid Q', L')}$$

EQUIV
$$\frac{P \equiv Q \qquad Q' \equiv P' \qquad \mathcal{P} \vdash (Q, L) \rightarrow (Q', L')}{\mathcal{P} \vdash (P, L) \rightarrow (P', L')}$$

LOGCALL
$$\frac{p \neq q}{\mathcal{P} \vdash (p :: [q] \cdot f([\tilde{a}]) \mid q :: e \mid Q, L) \rightarrow (p :: \mathsf{skip} \mid q :: e; [q] \cdot f([\tilde{a}]) \mid Q', L)}$$

CALLLOCK
$$\frac{\tilde{a}' = \mathcal{P}(f)_{\text{arg}} \qquad b = \mathcal{P}(f)_{\text{body}}[\tilde{a}/\tilde{a}'] \qquad \forall p, L_l(p) \cap L_r(p) = \emptyset \qquad L_r' = L_r[p \mapsto \emptyset] \qquad L_l' = L_l[p \mapsto L_r(p) \cup L_l(p)]}{\mathcal{P} \vdash (p :: [p] \cdot f([\tilde{a}]) \mid Q, L) \rightarrow (p :: \mathsf{lock}\ (L_r(p))\ b \mid Q', L')}$$

REQLOCKS
$$\frac{\text{need} = \tilde{a} - (L_l(p) \cup \{p\}) \qquad L_r' = L_r[p \mapsto \text{need}] \qquad L_l' = L_l}{\mathcal{P} \vdash (p :: [p] \cdot f([\tilde{a}]) \mid Q, L) \rightarrow (p :: [p] \cdot f([\tilde{a}]) \mid Q, L')}$$

LOCKSTEP
$$\frac{\mathcal{P} \vdash (p :: e \mid Q, L) \rightarrow (p :: e' \mid Q', L')}{\mathcal{P} \vdash (p :: \mathsf{lock}\ \tilde{p}\ e \mid Q, L) \rightarrow (p :: \mathsf{lock}\ \tilde{p}\ e' \mid Q', L')}$$

UNLOCK
$$\frac{L_l' = L_l[p \mapsto L_l(p) - \tilde{p}] \qquad L_r' = L_r \qquad e = \mathsf{skip} \vee e = [v]}{\mathcal{P} \vdash (p :: \mathsf{lock}\ \tilde{p}\ e \mid Q, L) \rightarrow (p :: e \mid Q, L')}$$

Table 3.1: SCOOP Rewrite Rules

call to the target handler, to be executed after the current tasks of the target handler. The caller proceeds without waiting. Recall that the notation $[\tilde{a}]$ to describe a fully evaluated sequence of expressions.

When a call is already on its target handler, either because it was moved there by its client, or because it is a non-`separate` call, the required locks must be requested. This behaviour is specified in REQLOCKS. Only the locks that are not already held by the client (need in the rule) are requested, and this includes the lock of the client itself.

Once the requests have been made the client must wait, as specified in CALLLOCK, until no other handler holds any of the required resources. Once the locks are available, they are transferred to the lock set of the client. The client's expression is rewritten from the call to the internal lock expression with the the body of the call with the arguments instantiated as its own body. The locks that were just taken are also present as the first component of the lock expression. This will allow just those locks to be later released. Here the notation $\mathcal{P}(f)_{\text{body}}[\tilde{a}/\tilde{a}']$ expresses the substitution of the actual arguments $\tilde{a}$ for the formal arguments $\tilde{a}' = \mathcal{P}(f)_{\text{arg}}$ within the body $\mathcal{P}(f)_{\text{body}}$ of routine f. In the previous two rules, the sequence of values $[\tilde{a}] = [p_1], \ldots, [p_n]$ is reinterpreted as a set.

Once the body of a routine is wrapped in the lock syntax, it has to be actually executed. To do this, the LOCKSTEP rule allows a lock expression to progress if the inner part can progress independently in the same state.

Lastly, once the inner part of the lock expression has finished, it can let go of

p	r_1	r_2	L_l	L_r
$[p] \cdot \texttt{run}([r_1], [r_2]); e'$	skip	skip	$\{p \mapsto \emptyset\}$	$\{p \mapsto \emptyset\}$
SEQSTEP and REQLOCKS are applied on p				
$[p] \cdot \texttt{run}([r_1], [r_2]); e'$	skip	skip	$\{p \mapsto \emptyset\}$	$\{p \mapsto \{r_1, r_2\}\}$
SEQSTEP and CALLLOCK are applied on p				
lock $[r_1, r_2]$ $([r_1] \cdot \texttt{f}; [r_2] \cdot \texttt{f})$	skip	skip	$\{p \mapsto \{r_1, r_2\}\}$	$\{p \mapsto \emptyset\}$
SEQSTEP, LOCKSTEP, and LOGCALL are applied on p				
lock $[r_1, r_2]$ $([r_2] \cdot \texttt{f}))$	skip; $[r_1] \cdot \texttt{f}$	skip	$\{p \mapsto \{r_1, r_2\}\}$	$\{p \mapsto \emptyset\}$

Figure 3.2: Execution derivation for deadlock example in Figure 3.1. Each column is either a processor (p, r_1, r_2) or a component of the state (L_l, L_r).

the locks that it holds. An expression that is finished is either a skip expression or a handler value, $[v]$. Recall that $\tilde{p}$ is only the set of locks that were not already held when the call was initially made, so revoking these locks will not revoke any locks that were held before the call was made.

Note that the lock-passing mechanism present in some SCOOP implementations is not modelled by this semantics.

Example 3 To illustrate the use of the rewrite rules, they are applied to Figure 3.1. System execution starts with a call `make` on an initial handler p. An execution step of the body of `make` is shown, demonstrating an application of CREATE$_1$ on the instruction `create x`:

$$(p :: \mathsf{create}(q); e \mid Q, L) \to (p :: e \mid q :: \mathsf{skip} \mid Q, L)$$

Here, assume that the handler tag of the local variable `x` is q and can be obtained with a mapping *tag* : $Name \to Tag$.

The other `create`-statements will give rise to more concurrent processes. Finally, the routine `run` is called, under the assumption that *tag*(`d1`) $= r_1$ and *tag*(`d2`) $= r_2$ to get the derivation in Figure 3.2. Here, ignoring the STEP rules for a moment, the first rewrite comes fromREQLOCKS and shows that the handlers of `d1` and `d2` are added to the request set of p. The second rewrite is performed according to the CALLLOCK rule, and shows that the requested locks (which are available, in this example) are taken by adding them to p's set of held locks. The last step is an application of rule LOGCALL and shows how an asynchronous call is transferred to its handler. Discussion of rules SKIP, SEQSTEP, and LOCKSTEP is omitted as they do not provide any further insight on this example.

3.3 Ensuring deadlock freedom

Having the syntax and semantics, a static approach to ensuring that a program is free of deadlocks can be defined. The first step in this process is to choose a fundamental property that will, in turn, guarantee an absence of deadlocks in the execution of the program. Of the Coffman deadlock conditions, the only one that can be reasonably provided is that no cycles exist in the lock graph. To guarantee that this property does not exist the problem is phrased in terms of partial orders, which through transitivity and anti-symmetry translates to acyclicity. This section describes how this property is expressed in routine interfaces, what a well-formed expression is, and proves that well-formed expressions ensure that a program is deadlock-free when executed via the semantics in Section 3.2.2.

3.3.1 Annotation language

At the class level, annotations of the following form are allowed:

$$\text{classHeader} ::= \texttt{class}\ \text{ident} \ \big|\ \texttt{class}\ \text{ident}\ \texttt{<}p(,p)^*\texttt{>}$$

A class can thus be parameterized with the handler tags it is using. This is just an extension of the idea of formal generic parameters, except for `separate` handler types, and provides a way to introduce handler variables. Consider Figure 3.3 for example, the class `DEADLOCK` has formal generic parameters `xp` and `yp`, which introduces these handler variables in the body of the class.

An instance is declared as follows:

```
d : DEADLOCK <p, q>
```

This entity, `d`, based on class `DEADLOCK` instantiates `xp` and `yp` with `p` and `q`. This mechanism allows uses the familiar generic mechanism for code reuse to be extended to the deadlock freedom annotations, meaning that clients can reuse the same classes more readily by instantiating them with different handlers.

The preconditions of routines represent the required ordering of handlers, expressed using the following syntax (note that the non-strict ordering symbol replaces the strict ordering symbol in the program text to make it easier to type, however the interpretation should remain non-strict in all cases)

$$\text{req} ::= \epsilon \ \big|\ \texttt{require}\ p < p(,p < p)^*$$

For example, in Figure 3.3, the routine `g` is annotated to express that the handler `yp`, which will be locked as a result of the execution of the body of `g`, is below handler `xp`, which is locked by calling `g`.

The interface of a routine also states the set of locks that may be taken during the execution of the routine body.

$$\text{ens} ::= \epsilon \ \big|\ \texttt{lock}\ p(,p)^*$$

For example, the `lock` clause of routine `f` in Figure 3.3 expresses that `x`'s handler, `xp`, may be taken by executing the body of `f`, as the call `g(x)` will lock this handler.

This expanded syntax also then modifies the representation of the interfaces as they are stored in the program. In particular, the routine specification is expanded to accommodate the annotation language as follows:

$$rtn \in Routine = \wp(Tag \times Tag) \times Tag^* \times \wp(Tag) \times Expr$$

and the named expansion of the components of this type is:

$$rtn = (rtn_{\leq}, rtn_{\vec{as}}, rtn_{\mathsf{locks}}, rtn_{\mathsf{body}})$$

```
class DEADLOCK <xp, yp>

feature
  x : separate <xp> S
  y : separate <yp> S

  f
    require
      yp < xp
    lock
      xp
    do
      g (x)
    end

  g (a : separate <xp> S)
    require
      yp < xp
    lock
      yp
    do
      h (y)
    end

  h (b : separate <yp> S)
    do
    end

  set (a_x : separate <xp> S;
       a_y : separate <yp> S)
    do
      x := a_x
      y := a_y
    end
end
```

Figure 3.3: Annotated DEADLOCK class

The component $rtn_{\leq}$, corresponding to programmer provided `require` annotations as in Section 3.3.1, is a relation on handler tags, describing the partial order on handlers required by the routine. The set of locks that may be taken as the result of executing the body of the routine is given by rtn_{locks}, corresponding to `lock`; this is the other programmer-provided annotation. The remaining components are as they were previously.

In Figure 3.3 g's interface requires that `yp < xp`. Due to the construction of the two deadlock variables, `d1` and `d2`, in Figure 3.1, there is a contradiction in the stated ordering of the handlers. Recall that `d1` and `d2` are constructed like so:

```
create d1.set (x, y)
create d2.set (y, x)
```

which means their types are actually:

```
d1: DEADLOCK <ap, bp>
d2: DEADLOCK <bp, ap>
```

given that `ap` is the handler of object `a` and `bp` is the handler of object `b`. Therefore the handler variables `xp` and `yp` are instantiated differently in `d1` and `d2`. However, this difference itself causes no problem. An annotated `MAIN` class is shown in Figure 3.4.

The problem arises through the calls to `f` in the body of `run`. It is here that the requirements on the routines interfaces come into conflict. To be precise: `d1.f` requires that `yp < xp` and `d2.f` requires `xp < yp`. These cannot be mutually satisfied unless `xp = yp`. If this is not the case then it is impossible to annotate `MAIN` from Figure 3.1 such that it can satisfy the well-formedness predicate coming next.

3.3.2 Well-formed programs

The scheme for ensuring that a program is well-formed ensures that there exists, for each routine, a consistent handler ordering (through $rtn_{\leq}$). Additionally, it

```
class MAIN <ap, bp>
feature
  x: separate <ap> S
  y: separate <bp> S

  run (d1: separate DEADLOCK <ap,
     bp>
       d2: separate DEADLOCK <bp,
     ap>)
    do
      d1.f
      d2.f
    end
```

```
  make
    local
      d1: separate DEADLOCK <ap,
    bp>
      d2: separate DEADLOCK <bp,
    ap>
    do
      create a
      create b
      create d1.set (a, b)
      create d2.set (b, a)
      run (d1, d2)
    end
ebd
```

Figure 3.4: Annotated `MAIN` class

ensures that locks are declared (rtn_{locks}) properly, and within the scope of these declared locks the callee's locks (rtn'_{locks} instantiated by its arguments) do not lose any of the knowledge that the declared locks are held. The well-formedness property of a program relies on every routine being well-formed:

$$\mathit{wfProgram}_{\mathcal{P}} = \forall rtn \in \mathit{range}(\mathcal{P}).\ \mathit{wfRoutine}_{\mathcal{P}}(rtn)$$

A well-formed routine must ensure that the relation specified in its interface is an order (first clause) and that the routine body is consistent with this order (second clause):

$$\mathit{wfRoutine}_{\mathcal{P}}(rtn) = \mathit{isOrder}(rtn_{\leq}) \wedge \mathit{wfExpr}_{\mathcal{P}}(rtn_{\leq}, rtn_{locks}, rtn_{body})$$

The well-formedness of expressions is therefore specified in terms of both the order of the routine that contains the given expression, and also the locks that the routine declares:

$$\begin{array}{lll}
\mathit{wfExpr}_{\mathcal{P}}(\leq, lks, \mathsf{lock}\ \tilde{p}\ e) & = & \mathit{False} \\
\mathit{wfExpr}_{\mathcal{P}}(\leq, lks, [p]) & = & \mathit{True} \\
\mathit{wfExpr}_{\mathcal{P}}(\leq, lks, \mathsf{skip}) & = & \mathit{True} \\
\mathit{wfExpr}_{\mathcal{P}}(\leq, lks, \mathsf{create}(p)) & = & \mathit{True} \\
\mathit{wfExpr}_{\mathcal{P}}(\leq, lks, e_1; e_2) & = & \mathit{wfExpr}_{\mathcal{P}}(\leq, lks, e_1) \wedge \mathit{wfExpr}_{\mathcal{P}}(\leq, lks, e_2) \\
\mathit{wfExpr}_{\mathcal{P}}(\leq, lks, t \cdot f(\tilde{a})) & = & \\
 & & \begin{array}{ll}
 (1)\ \mathit{inst}_{\leq} \subseteq \leq & \wedge \\
 (2)\ \forall a \in \tilde{a}.\ \ \mathit{wfExpr}_{\mathcal{P}}(\leq, lks, a) & \wedge \\
 (3)\ \forall a \in \tilde{a}.\ \ \mathit{tag}_{\mathcal{P}}(a) \leq \mathit{tag}_{\mathcal{P}}(t) & \wedge \\
 (4)\ \mathit{wfLevels}_{\mathcal{P}}(\leq, \mathit{inst}, lks, \tilde{a}) & \\
 \qquad \text{where}\ \ \mathit{inst} = \mathcal{P}(f)[\tilde{a}/\mathcal{P}(f)_{\vec{as}}] &
 \end{array}
\end{array}$$

The cases of values, skip, and create are always well-formed because they involve no locking of handlers. Sequential composition of expressions must guarantee that both the first and second component of the composition are well-formed under the same order and lock set.

Call expressions are the most intricate case as this is where deadlocks can arise in SCOOP. A call $t \cdot f(\tilde{a})$ must satisfy the following properties:

1. Order consistency – the order that the interface of f defines is consistent with the order of the caller, $\leq$.

2. Well-formed arguments – each argument expression must be individually well-formed. This is analogous to the sequential composition case.

3. Target before arguments – The order must express that the target of the call is locked before (i.e., is greater than) each argument.

4. Well-formed levels – This guarantees that the call will only lock things that are in the lock set (as defined by the surrounding routine), and that the new locks are all "less than" the lock set.

Further, the "well-formed levels" property is formalized as:

$$\mathit{wfLevels}_{\mathcal{P}}(\leq, \mathit{inst}, \mathit{lks}, \tilde{a}) = ((\mathit{inst}_{\mathit{locks}} \times lks) \subseteq \leq) \wedge (\mathit{tags}_{\mathcal{P}}(\tilde{a}) \subseteq \mathit{lks})$$

The first clause of *wfLevels* has all associations between the declared locks of the call and the context locks related by the context order relation. In other words, this states that each declared lock of a call must be less than all of the context-locks, so that the execution only locks "down" the partial order. Since a routine may have no arguments and still lock some handlers in its body we compare context-locks against the `lock` clause, and not the arguments. The second clause states that if a routine does have arguments, then these arguments must be a subset of the `lock` clause, for consistency.

Example 4 The following shows the evaluation of the predicate *wfExpr* on the call of the routine `g` in the body of routine `f` from Figure 3.3. To make the example more varied, assume that the argument $[xp]$ and the corresponding lock of g are replaced by $[zp]$.

Let $ord = \{(yp, xp), (xp, t)\}^*$, where R^* is the reflexive transitive closure operator. As $(xp, t) \in ord$ and $\mathit{inst}_{\leq} = ord$, the predicate is satisfied:

$$\begin{aligned}
&\mathit{wfExpr}(ord, \{xp\}, [t] \cdot g([xp])) = \\
&\quad (xp, t) \in ord \\
&\wedge\ \mathit{wfExpr}_{\mathcal{P}}(ord, \{xp\}, [xp])) \\
&\wedge\ \mathit{inst}_{\leq} \subseteq ord \\
&\wedge\ \mathit{wfLevels}(\{(yp, xp), (xp, t)\}, \mathit{inst}, \{xp\}, \{xp\})
\end{aligned}$$

The following facts about the interface of the routine g and the instantiation with the argument xp are also used:

$$\begin{aligned}
\mathcal{P}(g) &= (\{(yp, zp), (zp, t)\}^*, [zp], \{zp\}) \\
\mathit{inst} &= \mathcal{P}(g)[[xp]/[zp]] = (ord, [xp], \{xp\})
\end{aligned}$$

and that the predicate *wfLevels* is satisfied because values are well-formed, and (xp, xp) is in the order (reflexivity).

$$\mathit{wfLevels}(ord, \mathit{inst}, \{xp\}, [xp]) = (\{xp\} \times \{xp\}) \subseteq ord \wedge \mathit{tags}_{\mathcal{P}}([xp]) \subseteq \{xp\}$$

$$\textsc{callLock}\ \frac{L'_b(p) = (f_{\mathit{locks}}[\tilde{a}/\tilde{a}']) : L_b(p) \quad L'_{\leq} = (L_{\leq} \cup (f_{\leq}[\tilde{a}/\tilde{a}']))^* \quad \ldots}{\ldots} \qquad \textsc{unlock}\ \frac{L_b(p) = b : L'_b(p) \quad \ldots}{\ldots}$$

Table 3.2: Instrumented rules

3.3.3 Deadlock freedom

Intuitively, this technique ensures that there exists a global ordering for every well-formed program, and also that during the execution of a program each handler obeys an order in which to take locks. Deadlock-freedom follows from the fact that the acyclicity of the locking state is preserved under any execution step.

This approach, as mentioned, formalizes these ideas by building on the notion of the lock graph from [22]. The reasoning does not show directly that the rewriting the operational semantics cannot get "stuck" due to lock requests; this property follows from the satisfaction of the acyclical lock-graph property from the Coffman conditions. Translated to the SCOOP model a locking graph has handlers (resources) as nodes. There is an edge (p, q) in the graph if some client has locked handler p while requesting handler q. A locking-state L induces a locking-graph relation $\mathit{graph}(L)$ as follows:

$$\mathit{graph}(L) = \left(\bigcup_{p \in \mathit{dom}(L)} L_r(p) \times L_l(p)\right)^*$$

In this formalization of a lock graph, each handler p contributes some sub-graph which are combined for all handlers through the large union operator, and then the reflexive transitive closure is taken.

While the lock state L contains enough information to specify and detect when deadlock occurs, it does not have enough information to *prove* that the well-formedness conditions stated previously will guarantee that no cycles occur in the graph. Therefore, the lock state L is augmented with some extra "ghost" state which will record information during the execution of the program. These new pieces of state are: a lock-barrier $L_b : \mathit{Tag} \rightarrow (\wp(\mathit{Tag}))^*$ and a run-time ordering $L_{\leq} \in \wp(\mathit{Tag} \times \mathit{Tag})$. The lock barrier represents the stack of sets of upper bounds on the locks that may be requested, the stack follows the execution stack as calls are made. The run-time ordering is the ordering which is built up during execution. For the sake of the proof, the locking semantics has to be instrumented with these concepts. The minimal additions to the semantics are shown in Table 3.2. The essential behaviour is that when a new call is made the a new "barrier" is pushed onto the stack of lock barriers, and popped off when the call returns. Additionally, when new calls are made the run-time ordering is updated with that of the call's instantiated interface. These new pieces of data let the proof verify that the order is maintained.

With this extra information, the following property, *sound*, is proved invariant under execution. The predicate states that the run-time ordering $L_{\leq}$ is indeed a partial order, that the locking barrier is respected, and that the locking graph is acyclic.

$$\begin{array}{rcll} \mathit{sound}(L) & = & \mathit{isOrder}(L_{\leq}) & (1) \\ & \wedge & \forall p \in \mathit{dom}(L).\ \mathit{top}(L_b(p)) \times L_l(p) \subseteq L_{\leq} & (2) \\ & \wedge & \mathit{graph}(L) \subseteq L_{\leq} & (3) \end{array}$$

Here, *top* denotes the first element of a sequence.

Theorem 1 *Given a well-formed program $\mathcal{P}$ and an instrumented rewrite rule $\mathcal{P} \vdash (P, L) \rightarrow (P', L')$, sound$(L)$ implies sound(L').*

Briefly, the third clause is of primary concern; if the locking-graph ($graph(L)$) is a subset of an order, then it must be acyclic. Since $L_{\leq}$ is an order, thus acyclic, so is its inverse.

The initial two clauses support this goal, with the first establishing that as the program executes the relation that is specified piece-wise in the routine annotations is indeed an order. This fact follows from the definition of *wfRoutine* and the instantiation of the routines in the first clause of the call-case of *wfExpr*.

The second clause of *sound* states that the new upper-bound on locks is below all other locks that have already been acquired by the client p. The proof of this property is garnered from the Cartesian product in the *wfLevels* predicate, which imposes that when locks are taken, they are less than every lock taken by the surrounding procedure. When function calls are nested, these transitively combine to ensure that locks requested by a client p are less than all other locks currently held by that client. So, whenever locks are taken, they are "below" the currently held locks, and this is the essential property that proves the third clause.

3.3.4 Reducing annotations

Annotation burden is an important aspect of any technique which asks the user to provide extra information about the program. In an effort to reduce the annotation burden, two main sources of annotations are identified:

- Necessary annotations that are required because of the well-formedness rules.
- "Leaky" annotations that describe handlers that should be invisible outside the routine.

A "leaky" annotation would be something like,

```
foo <p>       -- p is a handler variable
  local
    x : separate <p> C
  do
    create x  -- creates x on handler p
    bar (x)   -- locks and uses x
              -- assume p is unused past
              -- this point
  lock
    p
  end
```

This routine requires that all calls to `foo` supply a handler `p` and must also make sure that `p` is in the callers lock set. Such a situation means that all specifications that involve `p` have to be copied up through the call stacks. Now, the assumption here is that `p` is unused outside of `foo` (although clearly `bar` uses it).

To reduce the annotation burden, these two problems have to be addressed. The first problem is addressed by adding the necessary annotations to a routine based on its body. This is done in phases, one for order information, and another for lock set information. To reconstruct order information, the following reconstruction is done for every call,

$$\mathrm{reconOrd}(t \cdot f(\vec{as}), \mathrm{ord}) = \mathrm{ord} \cup (t < \vec{as}) \cup f_{<}$$

accumulating the ord result and adding it to the order for the surrounding routine.

Likewise, reconstructing lock set information is performed by using

$$\mathrm{reconLks}(t \cdot f(\vec{as}), \mathrm{lks}) = \mathrm{lks} \cup \mathit{tags}\ \vec{as}$$

which collects all locks taken in the body of a routine and adds them to the lock set of the calling routine.

To reduce the instances of "leaky" annotations, declarations of handlers are allowed to be local to a particular routine. Using this in the previously leaky routine results in the following:

```
foo
  local
    <p>       -- p is a local handler
              -- variable
    x : separate <p> C
  do
    create x  -- creates x on handler p
    bar (x)   -- locks and uses x
              -- p cannot be used past
              -- this point
  end
```

Now the handler `p` is local to the routine. This also makes `p` implicitly part of the `lock` set, although it will not be visible there from outside the routine so the `lock` set is left blank here. Additionally, since it is local, it cannot leak outside the routine (i.e., by hiding it in another object), and any attempt to do so will cause an unbound handler variable error at compile time. Because it doesn't leak outside the routine, this means that callers of `foo` will not have to reason about or cooperate with `p` because they do not and can not know about it.

3.3.5 Evaluation

To evaluate the effectiveness of the deadlock approach presented here, it was applied to an example application. The application is a simple web server that is able to respond to basic `GET` requests from a client with static web pages. The server is limited to providing just successful (200) and file not found (404) responses, and serving from a local filesystem.

The server was shown to be deadlock free by the deadlock tool implementing the approach proposed in this work. However, in any approach which requires that annotations be added to a program, it is informative to examine what the annotation burden is, i.e., how many annotations are required for a given amount of code. Table 3.3 gives a breakdown of the lines of code, annotations required,

Class	Lines	Locks	Order	Over.	I-Locks	I-Order	I-Over.
`APPLICATION`	45	2	5	16%	0	1	2%
`HTTP_200`	55	2	0	4%	1	0	2%
`HTTP_404`	27	1	0	4%	0	0	0%
`HTTP_REQUEST`	45	0	0	0%	0	0	0%
`HTTP_RESPONSE`	41	0	0	0%	0	0	0%
`MIME_GUESSER`	72	0	0	0%	0	0	0%
`REQUEST_HANDLER`	103	2	2	4%	0	0	0%
Total	398	7	7	4%	1	1	0.5%

Table 3.3: Classes and annotation measurements

and overhead per module in the server. It also gives the annotations requried when the the locks and orders are inferred using the reconstruction algorithm. Two types of annotations are measured, annotations that specify what locks are taken and also which orders had to be specified. The totals are also listed, and the server only required 4% of its lines to be annotations to prove deadlock freedom. When the annotations are inferred, the overhead falls to 0.5%.

To get a better impression of the annotations that were used, Figure 3.5 shows a snippet of the requeset handling function. The only part that has to be stated is that the socket (hidden in the `RESPONSE` object) will be locked, and it is “less than” the current process (`dot`). These annotations can also be inferred when using the annotation reconstruction scheme.

This evaluation is limited to the concurrency patterns that could be revealed through the web server application. There are other patterns which are not evaluated, though this does show that the technique can be applied to a useful concurrent application.

3.4 Tool

This static technique has been implemented in a prototype tool, written in Haskell, and available from [104].

Haskell was chosen as the implementation language to match the mathmematical nature of the routine annotations, as they are either order relations or lock sets. Since Haskell makes immutible structures and pure functions the default, this reduced the possibility for errors to occur in the implementation via accidental mutation.

Due to this, the implementation of the well-formedness checks corresponds nearly directly to the mathematical description. Of course, some modification is necessary to propegate type errors in the Eiffel source that is being checked up to the top level to report.

Additionally, a GUI is implemented to make the interaction easier by providing graphical feedback when orders are violated. The expected order is presented

```
req (sock : separate <s> NET_SOCK)
  require
    s < dot
  local
    last     : STRING
    http_req : HTTP_REQUEST
  do
    create http_req.make ()

    from
      read_line (sock)
    until
      last.is_equal (cr)
    loop
      http_req.add_field (last)
      read_line (sock)
    end

    process_request (http_req)
  lock
    p
  end
```

Figure 3.5: HTTP request processing

in a graph beside the actual order so the discrepency can be more easily seen; layout is provided by the GraphViz tool.

3.5 Coq proof

This section describes a proof that a program which passes type-checking for the deadlock prevention scheme really will not deadlock.

The proof is contained in its own `Section`, used to encapsulate parameters and hypotheses. `Section` *Config*.

`Parameter` *Name* : `Set`.

Expressions. Expressions are either a value `ok p`, `skip`, a routine call, sequential composition, or a `lock` that cannot be written in the program, but serves as a mechanism to mark within the program expressions how many locks have been taken (lock depth). These represent a more restricted syntax, but capture the essence of basic SCOOP programs through the `call` and `lock` constructors.

`Inductive` *expr*: `Type` :=
| *ok* (*p*: *HandTag*): *expr*
| *skip*: *expr*
| *call* (*target*: *HandTag*) (*f*: *Name*) (*args*: *list HandTag*): *expr*
| *lock* (*lks* : *HandSet*) (*e*: *expr*): *expr*
| *seqc* (*e1 e2*: *expr*): *expr*.

For display `;;` is used as an alternative infix notation for `seqc`.

```
Notation ”e1 ;; e2” := (seqc e1 e2)
                              (at level 90, right associativity): core_scope.
```

Programs. A function entry is a combination of:

- the body, which is just an expression
- the locks that are allowed to be taken in the body
- and the sub-order that the function will use

This represents the annotations that allow us to reason about deadlock freedom. These are the “locks” set and “require” order for the given function.

```
Record func_entry : Type :=
  { func_body: expr;
    func_allowed: HandSet ;
    func_ord : HandRel
  }.
```

A program is a mapping from names to function entries. This will allow the call expressions to be interpreted by replacing their names with their bodies

```
Notation ”’program’” := (Name → func_entry).
```

The proof is parameterized both in the program that is being considered, and the order relation that will be used to determine that the locking order is correct.

```
Parameter P: program.
Parameter ord: HandRel.
Hypothesis is_order: HandOrder ord.
```

Convenient short hand notations for the allowed locks and body of a given function name.

```
Definition allowed (f: Name) := func_allowed (P f).
Definition body (f: Name) := func_body (P f).
```

Well-formedness. A well formed expression (one that passes type checking) can be constructed in many ways. This uses an inductive predicate to encode the type checking rules. It is a relation between a set of handlers that are allowed to be locked, and an expression.

```
Inductive WfExpr: HandSet → expr → Prop :=
```

An `ok` expression is immediately well formed

```
  | WfOk (a: HandSet) (p: HandTag) : WfExpr a (ok p)
```

The do-nothing operation, `skip`, is also well formed in all contexts.

```
  | WfSkip (a: HandSet): WfExpr a skip
```

The sequential composition of two expressions is not as the above two, but only requires that each subexpression be well formed with respect to the same set of allowable locks.

```
| WfSeq (a: HandSet) (e1 e2: expr)
          (WfE1: WfExpr a e1) (WfE2: WfExpr a e2):
     WfExpr a (e1 ;; e2)
```

A call is when locking occurs, so one would expect that describing that a well formed call is the most interesting case. In particular it demands that the locks to be taken, the arguments to the call, are a subset of those that are allowed to be taken. Additionally, it requires that the body of the call is well formed. This is not strictly required, because it is assumed that the program as a whole is well formed, but makes later induction proofs easier to state this here.

The last requirements are that the relation between the allowed locks and the locks that the new call declares as being allowed form a less-than relationship, and that this relationship is consistent with the global order.

This will later allow a proof on stacks of sets of locks that have a pair-wise less-than relationship.

```
| WfCall (lks a a' : HandSet) target f args
           (ArgsInAllow: !args! ⊆ a)
           (WfBody: WfExpr (allowed f) (body f))
           (AllowedLess: (allowed f × a) ⊑ func_ord (P f))
           (SubOrder: func_ord (P f) ⊑ ord):
     WfExpr a (call target f args).
```

The lack of a constructor for a `lock` expression is to indicate that `lock` cannot occur in the *text* of a well formed program. Execution can (and does) introduce `lock` expressions though. Also, a well formed call demands that the body is well formed. This is a simplification that helps with proofs that use induction over well formed expression terms.

It is a simplificaition because the whole program is assumed to only contain well formed bodies (well formed function entries).

```
Hypothesis wfProg: ∀ f, WfExpr (allowed f) (body f).
```

Processes. A configuration is a list of individual handler configurations. The handler configurations contain the handler tag, the current expression that is being evaluated, a list of sets of handler tags that represent a stack of allowable handlers to take locks from, and lastly a list of sets of handler tags that represent the actual locks taken.

The two lists of sets of handlers are not present in the actual execution model, but they are tracked here because the proof needs some information that is not present in the model. They are essentially “ghost” state as they do not influence the execution of the model, they just along for the ride.

```
Inductive config: Type :=
| EmptyConfig : config
| ConsConfig : HandTag → list HandSet → list HandSet → expr → config → config.
```

A more convenient notation is used for configurations that makes them easier to read and more like the initial notation in the non-mechanized semantics.

```
Notation ”( p , A , LK ) :: e” :=
  (ConsConfig p A LK e EmptyConfig)
    (at level 0, e at level 70).
Notation ”( p , A , LK ) :: e — Q” :=
  (ConsConfig p A LK e Q)
    (at level 0, e at level 70, right associativity).
```

Operational semantics. The execution type defines the operation semantics of the program by induction. This inductive type forms a homogeneous relation between a pairs of configurations and sets of locked handlers.

```
Inductive Execution : config → HandSet → config → HandSet → Type :=
```

Skip composed with an expression simply writes away without modifying any of the normal or ghost state.

```
  | SkipR :
      ∀ p A LK e Q L,
        ⊢ ((p, A, LK) :: (skip ;; e) | Q, L) ⇒ ((p, A, LK) :: e | Q, L)
```

If the first component of a sequential can be rewritten in isolation resulting in new state, then its composition can be rewritten with the second componennt being unaffected.

```
  | Seq :
      ∀ p A A' LK LK' e1 e1' e2 Q Q' L L',
        ⊢ ((p, A, LK) :: e1 | Q, L) ⇒ ((p, A', LK') :: e1' | Q', L') →
        ⊢ ((p, A, LK) :: (e1 ;; e2) | Q, L) ⇒ ((p, A', LK') :: (e1' ;; e2) | Q', L')
```

Executing a call requires locks to be taken. Since this proof does not prove progress directly, but only via the Coffman conditions on ordering, whether they are currently taken or not doesn't really matter.

This rule introduces the `lock` expression that normally does not appear in a user-written program.

Note that the locks that the function `f` declares as allowable now go on the allow stack, and the newly locked arguments go on the lock stack.

```
  | TakeLocks:
      ∀ p Q L LK A target f args,
        ⊢ ((p, A, LK) :: call target f args | Q, L) ⇒
            ((p, allowed f :: A, (!args!) :: LK) :: lock (!args!) (body f) | Q, L)
```

The body of the lock can make progress while the lock is taken, this is the way executing a program under a lock is modeled. Much like the sequential composition operation, if the inner component of the lock expression can take a step, then the whole expression can move forward.

```
  | LockStep :
      ∀ p A A' LK LK' Q Q' L L' lks e e',
        ⊢ ((p, A, LK) :: e | Q, L) ⇒ ((p, A', LK') :: e' | Q', L') →
        ⊢ ((p, A, LK) :: lock lks e | Q, L) ⇒ ((p, A', LK') :: lock lks e' | Q', L')
```

When the body of the lock finally completes the lock expression can be removed. The completion of the body is indicated by the expression inside lock being `skip`.

```
  | LockPop :
      ∀ p a A l LK Q L lks,
        ⊢ ((p, a :: A, l :: LK) :: lock lks skip | Q, L) ⇒
        ((p, A, LK) :: skip | Q, L)
where "—- ( C , L ) ==¿ ( C' , L' )" := (Execution C L C' L').
```

Auxilary lemmas. This property of an expression merely proves if a given expression has a `lock` subexpression somewhere.

```
Inductive HasLocks: expr → Prop :=
  | HLLock: ∀ e lks, HasLocks (lock lks e)
  | HLSeq: ∀ e1 e2, HasLocks e1 ∨ HasLocks e2 → HasLocks (e1 ;; e2).
```

A useful property to have is that an expression only contains locks in the left part of sequential compositions. This means that any given expression only has descending chain of `lock` expressions, and that chain is the in the first component of sequential compositions.

This property allows reasoning on the depth of locks as it relates to what is in the stack of locks and allowable locks.

```
Inductive OnlyLeftLocks: expr → Prop :=
  | OLLOk: ∀ p, OnlyLeftLocks (ok p)
  | OLLSkip: OnlyLeftLocks skip
  | OLLCall: ∀ target f args, OnlyLeftLocks (call target f args)
  | OLLSeq: ∀ e1 e2, OnlyLeftLocks e1 → ¬ HasLocks e2 →
                                        OnlyLeftLocks (e1 ;; e2)
  | OLLLock: ∀ e lks, OnlyLeftLocks e → OnlyLeftLocks (lock lks e).
```

`lockDepth` is just a recursive function that will be used to measure how many locks descend down the left side of an expression.

```
Fixpoint lockDepth (e: expr): nat :=
  match e with
    | lock _ e' ⇒ 1 + lockDepth e'
    | e1 ;; e2 ⇒ lockDepth e1
    | _ ⇒ 0
  end.
```

To conduct the proof, there has to be an elaborated idea of a well-formed expressions. In particular, it must account for a wellformed `lock` expression.

The `StackWf` inductive type fills this gap, and to do this it makes use of the stack of allowable locks and already taken locks, in addition to the expression.

```
Inductive StackWf : list HandSet → list HandSet → expr → Prop :=
```

The `ok` and `skip` expressions are still trivially allowed.

```
| StkOk : ∀ A LK p, StackWf A LK (ok p)
| StkSkip : ∀ A LK, StackWf A LK skip
```

Sequential compositiion has become slightly more complicated. This is because the well-formedness must now account for the entire stack of locks, and it must know how much of the stack will be gone by the time second part of the composition, `e2`, is encountered. This amount is known, however, because it is just the number of lock expressions contained within the first part of the composition. Therefore, `e2` must be still be consistent when those locks are popped off the stack (through `skipn` on the stack) and that is expressed by the `WfE2` proof.

```
  | StkSeq A LK e1 e2
            (WfE1: StackWf A LK e1)
            (WfE2: StackWf (skipn (lockDepth e1) A)
                           (skipn (lockDepth e1) LK) e2):
     StackWf A LK (e1 ;; e2)
```

A call is `StackWf` if the top of the stack of allowable locks can ensure the well-formedness of the call. Again the body being well-formed is essentially included to make induction proofs easier, though it could be derived.

```
  | StkCall (a: HandSet) A LK target f args
             (WfBody: StackWf (allowed f :: nil) (!args! :: nil) (body f))
             (WfCall: WfExpr a (call target f args)):
     StackWf (a :: A) LK (call target f args)
```

A lock is well formed with regards to the stacks if its body is well formed with respect to the same stacks.

```
  | StkLock l A LK e (StkE: StackWf A LK e):
     StackWf A LK (lock l e) .
```

Two other simple properties that describe the consistency of the stack of allowable and taken locks are needed.

The first is that the stacks of locks are ordered with respect to the given ordering. This means that the locks at the top of the stack are "less than" than the locks immediately preceding the top. This basically forms the stack into layers of handlers, where the handlers in one layer are less than the layer above it.

```
Inductive OrdList : list HandSet → Prop :=
  | OLNil : OrdList nil
  | OLSingle : ∀ x, OrdList (x :: nil)
  | OLCons : ∀ x y l, x × y ⊑ ord → OrdList (y :: l) →
                             OrdList (x :: y :: l) .
```

The second property is that stacks of locks that are allowed to be taken and the locks that are taken must be in a pair-wise "less than" relation. This means that the handlers at the top of the allowed stack must be less than all the locks at the top of the locked stack. The stack below each of these must also have the same relation, this allows consistency as the locks are pushed and popped off.

```
Inductive LessLists: list HandSet → list HandSet → Prop :=
  | LLNil : ∀ A, LessLists nil A
  | LLCons : ∀ a A l LK, a × l ⊑ ord → LessLists LK A →
                                        LessLists (l :: LK) (a :: A) .
```

Soundness. A sound configuration the key property that should be maintained across steps in the operational semantics.

```
Inductive Sound : config → Prop :=
```

Empty configurations can't go wrong.

```
  | SoundEmpty : Sound EmptyConfig
```

If a configuration is non-empty, then it has to maintain the stack wellformedness of the expression with relation to the taken locks and the allowable locks.

Additionally, it must ensure that the taken locks and allowable locks are ordered lists, and that the allowable locks are always "less than" the taken locks. These properties are the heart of the proof, they directly give the Coffman condition that requested locks (those at the top of the allows stack) are always less than all of the taken locks.

The `OnlyLeftLocks` property is also required as a kind of sanity check on the execution.

```
| SoundCons
    p Q e A LK
      (StkLks: StackWf A LK e)
      (OrdAllows: OrdList A)
      (OrdLocks: OrdList LK)
      (LessPair: LessLists LK A)
      (OnlyLeft: OnlyLeftLocks e)
      (SndQ: Sound Q):
      Sound (ConsConfig p A LK e Q) .
```

The preservation property that should be maintained is the following, which states if a starting configuration is sound, then after making a step it is still sound. Since sound implies the absence of cyclical locking, the Coffman condition is satisfied after every step of the execution. Note that soundness here only depends on the configuration, not the state as it did in Theorem 1. This is because for the purposes of the proof, a lot of this information was moved into the configuration because this binds it more closely to the handler. Otherwise one must always look-up and update the locks and requests in the state, which is a more cumbersome arrangement for the proof. This is not a fundamental change from Theorem 1, the extra information has just moved closer to where it is used.

```
Lemma soundness L L' (C C': config)
  (Snd: Sound C)
  (Step: ⊢ (C, L) ⇒ (C', L')):
  Sound C'.

End Config.
```

This proof is a mechanically checked version of the informal proof sketch given previously. It shows the steps required to reason about the core of the deadlock freedom work, and provides a strong assurance than they are sound rules.

This can also be the starting point for more full proofs of the system, and even other properties of the SCOOP language which would complement the work on executable operational semantics by Morandi et al. [73].

3.6 Related work

The problem of describing, detecting, and preventing deadlocks in concurrent systems has spawned research based on a variety of approaches. Necessary conditions for a deadlock to occur have been described in a seminal work by Coffman et al. [22].

Dynamic techniques can be used to detect deadlocks, e.g. using techniques such as those presented by Bensalem et al. [7]. The fundamental approach in this work is to instrument the program and use this run-time locking information to detect locking cycles. The benefit is that this technique can be less conservative than the present approach, but it is based on actual program traces, and the results are, therefore, not sound.

Static techniques rely on programmer annotations to indicate a partial order among the program's locks, and statically check whether this order is abided by; this general idea is also the basis of the present approach. Korty [53] proposed a Lint-like tool for detecting deadlocks in programs with semaphores, however without soundness guarantees. Extended static checking for Modula-3 [27] and Java [33] uses program specifications in the style of Eiffel [67], from which verification conditions are generated and checked with an automatic theorem prover. Warnings are provided for various program errors, including deadlock. Being based on Eiffel-style specifications, annotations in this work are similar to this scheme. However, no soundness guarantees are given whereas this work guarantees deadlock-freedom for well-formed programs. Jacobs et al. [43] also generate verification conditions for annotated programs, and guarantee deadlock-freedom for programs verified with a static checker. In contrast to this work, they use a programming model for Java-like languages which is very different from SCOOP, and do not provide a rigorous formal locking semantics.

A number of static approaches to deadlock prevention are based on *type systems*, in particular using ownership types [20]. Boyapati et al. [14] have introduced the ability, as in this work, to create a directed acyclic graph, well-order, or tree to represent the underlying partial order. In contrast to this approach, the deadlock prevention scheme in the present thesis makes it possible to declare locking orders in a routine-local manner, which allows for a finer-grained modularity.

The present work distinguishes itself from the above approaches in that it has a higher-level concurrency model, not based on traditional threads, and thus has a coarser-grained locking behaviour.

Using a model similar to SCOOP, Kerfoot et al. [50] use types to ensure deadlock freedom for active objects [55]. Ownership types impose a hierarchy on active objects, but the variety of ownership-structures that are permitted are limited. Only trees are allowed, where the present approach can support a general directed acyclic graph. Ostroff et al. [79] develop a partial operational semantics for SCOOP, and consider liveness properties of programs in the context of model checking. While the approach can detect deadlocks, it is not a goal of the work to be a modular approach, thus it is doubtful that it would scale to large programs. Additionally, as it requires model checking, which is typically bounded, it can only provide sound results up to that bound. Kobayashi [52] gives π-calculus a type system that is able to infer and verify deadlock properties about a program. It gives a versatile approach that is even able to reason about recursive processes. However, the present work targets a new model of computation that is more immediately amenable to traditional imperative programming.

3.7 Conclusion

This deadlock freedom technique includes a static method for deadlock prevention in SCOOP. The model supports reasoning about deadlock well, as lock acquisition and release are related to routine invocation and return. This allows the annotations to be attached to the interface of routines, facilitating modular (per-routine) proofs of correctness. This aspect is essential in practice as it is easier to reason about deadlock when it is assured that local changes will not affect the overall result. An implementation of the scheme is available, and has been successfully applied to the example of a web server written in SCOOP.

Adding a deadlock prevention technique for SCOOP removes a critical deficiency of this particular model, but the results also provide important general lessons learned. While sound and scalable programming models for concurrency are overdue, the divide between formally driven language developments (such as process calculi) and concurrent programming language design still seems to be large. This work showcases how one may bridge this gap by using formal reasoning to derive techniques that can be applied to practical programming languages.

Additionally, this technique is a step towards increasing the confidence in SCOOP programs. A correctness property is proven for programs by using the type system to express and check that only non-deadlocking programs can be written.

A portion of this work has been featured and published at ICFEM 2010 [105].

Chapter 4

Testing of programs with contracts

There are some properties of programs that are more costly to ensure than others. Type checking is an inexpensive way (in terms of time) to verify simple properties. More complicated properties, however, may be too expensive to verify rigorously. Testing fits in these cases where some assurance of correctness is required but the cost of a proof is too great.

Therefore, it is important to have good testing techniques in the development of concurrent software as well as sequential. Such techniques difficult to provide, however, due to the unpredictable and irreproducible nature of thread scheduling, which makes interference between threads very difficult to discover. These difficulties have not stopped successful research into concurrent testing tools, e.g. [36, 94, 74, 54, 100]. It is also important that a technique have some sympathy for the typical development process. In particular, that developers may be only working on a small part of a large system. In such a case, testing the pieces in isolation is very important, typically using stub and mock objects to reduce dependencies.

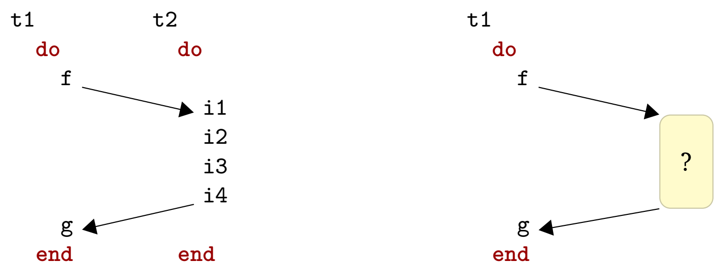
Attaining modularity and reproducibility in concurrent testing is difficult, as concurrent programs appear to be inherently non-modular and non-deterministic. The focus of this work is to address the difficulty in finding concurrency bugs, while keeping the process reproducible and modular.

Demonic testing is a new approach to solving these problems that combines contracts and other invariants from classes with advanced reasoning tools. This combination enables concurrent tests to be reproducible by "stubbing out" the other threads and replacing them with a reasoning engine. This helps to ensure the correctness of concurrent programs by making the testing process deterministic. Section 4.1 presents the underlying premise of demonic testing, an overview of the approach and introduces a running example. The entire testing method is described in detail in Section 4.2, including the foundational concepts and the translation of classes into the language of the supporting demonL tool. Section 4.3 shows how demonic testing is easily instantiated to test SCOOP programs. Section 4.4 gives a description of demonL and a formalization of how it works by translation to the Yices SMT solver. The technique is evaluated in Section 4.5. Discussion on related work follows in Section 4.6, and Section 4.7 concludes.

4.1 Premise and overview

One way to think about how concurrent programs execute is to take independent threads of control and interleave their actions in a global view of the system. This is how most operational semantics, like the one in Chapter 3, model the independent actions of programs that execute in parallel. For example, if one thread executes routine `t1` while another thread executes routine `t2` concurrently.

Figure 4.1a shows a simple example of this (edges are only there to indicate "context switching" to another thread), where `t2` could actually cause an atomicity violation or data race relevant to `t1`. If the objective is to *test* `t1` in isolation,



(a) Interleaving threads (b) Modularizing threads

Figure 4.1: Actual and desired thread interactions, edges indicate "context switches".

then somehow `t2` has to be removed. However, clearly just removing `t2` isn't useful: this approach would find no concurrency bugs. `t2` has to be replaced with *something*, visualized in Figure 4.1b. Keeping in mind that the goal is to find concurrency bugs in the execution of `t1`, the design of a replacement for `t2` must:

1. have a definition of what an error is, and
2. have a definition of what a feasible schedule is, and
3. provide a schedule that will cause the error from (1) and fits the constraints of (2).

If the missing part in Figure 4.1b can do these things, then it can provide a schedule which will drive the program to a failing state. For example, searching for a bug in Figure 4.1a one assumes the program state after the execution of `f` is acceptable, and upon `t1` resuming before `g`'s execution, it is failing because of the actions of `t2`. However, such an interleaving is only one of many, and these many interleavings present two problems for testing:

1. the interleavings are possibly exponential in the size of the program, and
2. they are unpredictable and some interleavings are more common than others, so finding the interleaving that leads to a fault is difficult.

This work proposes demonic testing, a technique combining program contracts with unit-testing. The demonic testing technique uses a purely logical description of system, extracted from the program contracts. The contracts are used to both define what an error is and what constitutes a feasible schedule. With this information, demonic testing then generates a schedule of operations which would drive the program to a fault. *Software Testing and Analysis: Process, Principles, and Techniques* [83] suggests that the costs of formal reasoning, in terms of devising the appropriate program invariants and proofs, can be balanced by using hybrid techniques. Demonic testing is such a hybrid technique that combines testing and symbolic state exploration. The basic demonic testing interaction can be seen in Figure 4.2. The program state after the execution of `f`

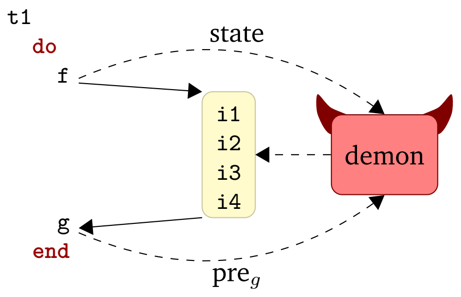


Figure 4.2: Demonic testing generating interference from the state and contracts. Solid edges represent the sequence of actions which would cause a fault, dashed edges represent the testing process.

and the precondition of the routine `g` are fed into a little "demon" generates a sequence of actions (`i1`, `e2`, `e3`, `e4`) that would brings the state after `f` to violate the precondition of `g`. Demonic testing repeats this procedure for every program point in a routine under test. In other words, demonic testing uses the *program's* notion of correctness to define an error in a concurrent setting, rather than the *language's* definition of correctness (i.e., do not dereference null) as is done in tools such as ConMem [111].

4.1.1 Overview

Looking more closely at the high-level description of the demonic testing process, some architectural decisions need to be made. The technique operates on the state of the program and the written contracts, and these two pieces are used by the "demon" to find buggy schedules. Demonic testing homogenizes these two pieces by working on only logical definitions.

The "demon" requires a way to interpret the contracts. For example, what does it mean for a list to be non-empty (`not list.is_empty`)? To do this, demonic testing uses the postcondition of `is_empty`, along with a fixed interpretation of `not`. The interpretation of an expression is therefore a combination of the postconditions of the program's functions and built-in interpretations of the common arithmetic and logical operators. The collection of all relevant program functions and operations is known as the *domain*. Also, since the dynamic state of the program is used, it has to be extracted and converted to this logical language.

With these requirements in mind, several processing steps must take place. Figure 4.3 shows a complete overview of the demonic testing process. Given a set of classes and one of their routines s to test, demonic testing must perform:

- **Class-to-domain transformation**: collect all supplier classes for the routine s, and convert them to a *domain description* for the "demon".
- **Routine instrumentation**: instrument the routine s to serialize the state and send it to the "demon" for analysis.

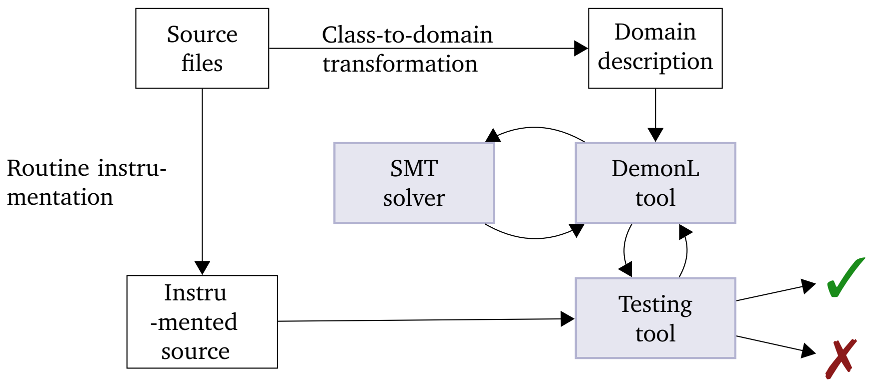


Figure 4.3: Overview of the system architecture

These two steps produce a *domain description* and an *instrumented routine* which are passed to the remaining two modules of the system:

- **DemonL tool** This component is the “demon”. It takes the domain description and the state as input and produces a schedule (sequence) of interfering actions.
- **Testing tool** This component runs the instrumented version of s with test cases. Test cases may for example be obtained from an automatic testing tool, such as AutoTest [68] or previously written unit tests.

If the demonL tool finds interference for a test case, the interfering instructions are given and the test fails. If no such interference can be found then the test succeeds.

An evaluation of the technique shows that it can successfully catch 7 out of 8 selected bugs of the concurrency bug collection [23], which include known bugs in major applications, such as Apache and MySQL – entirely without threads. The implementation of the technique is available [32, 102].

4.1.2 Example

To provide intuition for the demonic testing technique, this section introduces a running example.

The technique works with any language; contracts are required, but they can be embedded in comments as is done with JML [56] and ACSL [6], or as a domain-specific language, as with C# and .NET Code Contracts [21]. Languages that include contracts as first citizens include D [1] and Eiffel [67]. To demonstrate the technique, the implementation uses Eiffel as its source language, although it can be similarly applied to any language with contracts.

Example 5 The Eiffel class `IDLE_COUNTER` in Figure 4.4 represents the number of idle workers in the system. `IDLE_COUNTER` can increase and decrease the number of idle workers. The `wait_for_idle` routine decreases the number of idle workers if there are any, otherwise it waits on a condition variable until there are more idle workers.

```
class IDLE_COUNTER
feature
  num_idlers: INTEGER

  increment
    do ...
    end

  decrement
    require
      non_zero: num_idlers > 0
    do ...
    ensure
      num_idlers=old num_idlers-1
    end
```

```
wait_for_idle
  do
    if num_idlers = 0 then
      mutex.lock
      if num_idlers = 0 then
        -- release mutex and
        -- wait
        non_zero.wait (mutex)
      end
      mutex.unlock
    end

    decrement
  end
```

Figure 4.4: Work distribution example

Demonic testing uses the state at a given program point during execution to statically determine whether concurrent interference at that point could cause a fault. A *fault* exists if the next instruction to be executed could have its precondition violated due to other threads modifying shared state. The output of demonL is a sequence of instructions that would move the program into a state that would cause a fault. These instructions represent interference from another thread that give rise to a fault. For example, in Figure 4.2 the state after `f`, when manipulated by calling `e1`, `e2`, `e3`, `e4`, in order results in the precondition of `g` being violated.

Example 6 While running a unit-test of the `wait_for_idle` routine in Figure 4.4, the tool instruments the routine `wait_for_idle` with calls to the synthesis tool at every program point. This would change the last instructions of `wait_for_idle` to be:

```
wait_for_idle
  do
    ...
    end

    demonL (state, pre_decrement, rely_true)
    decrement
  end
```

This makes a call out to demonL with the state, the precondition of `decrement` and another parameter which is explained shortly. For a given test case, the tool reports whether or not interference could be found that will lead to a failure of the routine. In this example, the tool reports that an extra call to `decrement` immediately before the existing call to `decrement` will cause a violation.

There are two ways to respond to a warning produced by this method: either to modify the behaviour of the program so that it is not vulnerable to this kind of interference, or to refine the specification and only allow certain interference.

In the case of Figure 4.4, synchronization instructions could be introduced to prevent concurrent access of the shared data.

Example 7 The developer can express that the interference found in Example 6 for Figure 4.4 does not occur by limiting the interference that can be generated. For example, the restriction could be:

```
num_idlers >= old num_idlers
```

Demonic testing currently uses a specially tagged postcondition to specify this restriction. Using this postcondition would make the end of the `wait_for_idle` routine:

```
wait_for_idle
  do
    ...
    end

    demonL (state, pre_decrement, rely_condition)
    decrement
  ensure
    rely: num_idlers >= old num_idlers
  end
```

This special "rely condition" is incorporated into the call to demonL, limiting what it can generate for interference. Precisely, this rely condition disallows the environment from removing idle workers, implying that two threads cannot call this routine concurrently. The actual implementation has a master thread that is the only thread to call this routine, so the condition is satisfied by the implementation. The tool reports no violations when retesting the routine with this updated specification.

4.2 Demonic testing

This section presents the founding concepts and implementation of demonic testing as well as the approach to handling common synchronization primitives in a thread-free and modular way.

Recall that demonic testing takes classes annotated with traditional contracts and rely-specifications and uses static analysis in combination with the state from run-time to indicate where there may be errors due to concurrent executions.

4.2.1 Application of rely-guarantee reasoning

The rely-guarantee formalism [45] provides a framework to express and reason about interference in concurrent programs. The interaction between a component and its environment is included in the component's specification, allowing compositional reasoning about concurrent programs.

The formalism proposes an extension of the usual Hoare logic specification $\{P\}\ s\ \{Q\}$ to $\{P, R\}\ s\ \{G, Q\}$, which additionally contains a *rely*-condition R and a *guarantee*-condition G. The new conditions are binary predicates on states and describe the state changes that the environment (other threads) is allowed to make. A program s satisfies its specification if, starting in a state satisfying

P, under environmental interference adhering to R, s only makes state changes allowed by G, and finishes in a state satisfying Q.

The demonic testing approach uses a subset of the rely-guarantee concepts, namely the *rely*-conditions and the notion of *stability*, to specify interference generation for concurrent programs. The rely-specifications are manually added to the method under test, expressed as a postcondition with the tag `rely`. The `rely` tag indicates that this is only for demonic testing.

Stability is a definition of whether a routine can operate correctly in spite of the interference described by the rely-specification. Formally, the stability of a state-predicate p with respect to a rely-condition R is given as:

$$stable(p, R) \equiv \forall \sigma, \sigma'.\ p(\sigma) \land R(\sigma, \sigma') \to p(\sigma')$$

With the notion of the rely-condition one can express the goal of the testing strategy in the following terms: given the rely-condition R of a routine s under test, try to create interference that would drive the program to violate the precondition pre of some call in the body of s.

Example 8 In the running example, the following stability formula relates to the precondition of decrement:

$$num\_idlers(\sigma(this)) > 0 \land num\_idlers(\sigma'(this)) \geq num\_idlers(\sigma(this)) \to \\ num\_idlers(\sigma'(this)) > 0$$

Demonic testing distinguishes itself from other techniques of program verification by the usage of a dynamic program state to reduce the need for program specification. The goal given to the demonL tool is merely the negation of the $stable$ predicate, $\exists \sigma, \sigma'.\ p(\sigma) \land R(\sigma, \sigma') \land \neg p(\sigma')$. In demonic testing, this is formula simplified by:

1. assuming that the routine is correct without interference, and
2. only evaluating it on the test-cases that drive the routine.

The first point allows us to assume $p(\sigma)$, the second allows us to remove σ as a quantified expression, as it is given by the dynamic state. This leaves solving only $\exists \sigma'.\ R_\sigma(\sigma') \land \neg p(\sigma')$, where R_σ is the rely condition specialized to the concrete program state. Since the rely condition is specialized, it doesn't have to handle cases that never arise in normal program execution; this lowers the amount of required annotation. Additionally, there is no specification required for typically difficult to specify cases, such as loop variants and invariants.

4.2.2 Class transformation

An input (Eiffel) class C has three components: C_{name}, C_{attrs}, and C_{routines}. C_{name} denotes the name of the class. The attributes of the class, C_{attrs}, are denoted by $a : t$ to indicate an attribute a that has type t. Every routine s in C_{routines} has a name, denoted by s_{name}. Also, every routine can have a pre- and postcondition, denoted by s_{pre} and s_{post}.

The translation function to convert class files into demonL domains (see Figure 4.3) is shown in Table 4.1. Note that the presentation of this translation function uses a pattern-matching style, with the function matching arguments in a top-down fashion.

$\mathit{trans}(C)$	$=$	$\{\mathit{feat}(C, f) \mid f \in C_{\mathsf{features}}\} \cup \{\mathit{struct}(C)\}$
$\mathit{struct}(C)$	$=$	**type** C_{name} $\{$ C_{attrs} $\}$
$\mathit{feat}(C, f)$	$=$	$f_{\mathsf{name}}(\mathit{args}(C, f_{\mathsf{args}}))$ **require** $\mathit{expr}(f_{\mathsf{args}}, f_{\mathsf{pre}})$ **ensure** $\mathit{expr}(f_{\mathsf{args}}, f_{\mathsf{post}})$
$\mathit{args}(C, \vec{as})$	$=$	$(\mathit{this} : C_{\mathsf{name}}) :: \vec{as}$
$\mathit{expr}(\mathsf{args}, x.f(\vec{as}))$	$=$	$f(\mathit{expr}(\mathsf{args}, x), \mathit{expr}(\mathsf{args}, \vec{as}))$
$\mathit{expr}(\mathsf{args}, e_1 \mathbin{\mathsf{op}} e_2)$	$=$	$\mathit{expr}(\mathsf{args}, e_1) \mathbin{\mathsf{op}} \mathit{expr}(\mathsf{args}, e_2)$
$\mathit{expr}(\mathsf{args}, \mathsf{op}\ e)$	$=$	$\mathsf{op}\ \mathit{expr}(\mathsf{args}, e)$
$\mathit{expr}(\mathsf{args}, v)$	$=$	$\begin{cases} v & \text{if } v \in \mathsf{args} \\ \mathit{this}.v & \text{otherwise} \end{cases}$

Table 4.1: Translation function

- Attributes, along with the class name, are transformed into a data-type in demonL.
- Routines are transformed using *feat* directly into demonL procedures with pre- and postconditions.

The result of a function is denoted by having equality on the `Result` value, for example `Result = 2 * x`. Argument-list transformation of routines and functions explicitly includes the normally implicit self-reference in object-oriented programs. The translation of expressions is largely straightforward, with the target of a call moving to the first argument of the call, to coincide with the argument-list transformation. The translation doesn't treat inline creation expressions if they exist in pre or postconditions.

4.2.3 Routine instrumentation

As part of the technique, the routine under test must be instrumented (see Figure 4.3). The instrumentation augments the program execution so it is able to encode the dynamic state of the program for demonL. This is done by taking the available state from the `Current` variable, local variables, and the arguments to the instrumented routine and using reflection to serialize their contents. The current simple instrumentation strategy is to insert these calls at every program point, for the current state and next correctness condition. The correctness condition can either be a precondition, expression from a check instruction, or a postcondition. Figure 4.5 gives an example of the various cases. One can see that the rely condition is the same in every execution because it is a property of the *other* threads in the system, thus this will not change as the execution of the routine under test proceeds.

4.2.4 Handling synchronization primitives

The use of threads to construct concurrent programs inherently exhibits two types of effects:

- the *necessary*, where a thread contributes a result to another thread, and

```
t1
  do
    demonL (state, pre_f, rely_condition)
    f

    demonL (state, pre_g, rely_condition)
    g

    demonL (state, check_expr, rely_condition)
    check expr end

    demonL (state, post_expr, rely_condition)
  ensure
    rely: rely_condition
    post: postcondition_expr
  end
```

Figure 4.5: demonL instrumentation example

- the *incidental*, which are side-effects of necessary actions, and are also modifications to shared state.

When considering concurrent applications as a combination of necessary and incidental effects, the necessary aspect of concurrency can be seen as a dependency, and the incidental aspect can be seen as interference. One thread depends on another to provide a computational result in a shared memory location. In threaded programs, these dependencies are made explicit by a mutex's `lock`, or a condition variable's `wait` routine.

When unit-testing a class or method, it is common to provide stub methods or objects in the place of dependencies. For example, a full database connection may be replaced with one containing only a small fixed selection of data.

Although mutexes, semaphores, and condition variables carry no explicit invariants, their usage in programs is almost always accompanied by an implicit invariant related to a resource. Consequently, they can have meaningful post-conditions that can be used to create stubs to test concurrent programs without requiring threads. They merely need to be replaced with normal function calls that ensure the same postcondition.

Example 9 Assume a simple producer/consumer-style program, such as that given in Figure 4.6. The call to `cond_var.signal` in the `produce` routine has the precondition that the number of products is greater than zero. The counterpart in the `consume` routine, the call to `cond_var.wait`, has the same post-condition: `product > 0`.

To create a stub for the call to `cond_var.wait`, replace the implementation of `wait` on the condition variable with the one found in Figure 4.7. The new `wait` satisfies the invariant for the condition variable, and requires no other thread to work. The corresponding stub for `signal` would similarly have `product > 0` as a precondition and an empty body.

```
produce
  do
    product := product + 1

    if product = 1 then
      cond_var.signal
    end
  end
```

```
consume
  do
    if product = 0 then
      cond_var.wait
    end

    product := product - 1
  end
```

Figure 4.6: Producer/consumer coordination

```
wait
  do
    product := product + 1
  end
```

Figure 4.7: Example wait replacement

4.3 Applying demonic testing to SCOOP

Demonic testing, not being restricted to only detect data-races, can detect atomicity, order, and other kinds of violations if they are indicated by violating a precondition.

Although data-races are not possible in the SCOOP model, the other problems can still exist. To trigger an atomicity violation in SCOOP a client makes an assumption on the state of a handler's object after releasing control of the handler/object. Figure 4.8 shows how this may happen: a shared counter `c` is decremented after knowing that the count is at least 2, then decremented again. The `decrement` operation has a precondition that the count does not go below 0. The program in Figure 4.8, however, does not retain control of `c` and thus it may be that another thread decrements `c` between the `wait_decr` and `just_decr`. This will mean that the `c.decrement` call in `just_decr` will throw an exception.

4.3.1 Specializing demonic testing

Since SCOOP has guarantees that prevent data races it is possible to cut down on the complexity of the rely conditions in many cases. In particular, when an object is controlled then it cannot be the subject of external interference. Therefore, the default demonic testing approach must be modified slightly to account for this change in semantics. This can be done in one of two ways, by modifying the rely conditions or changing demonL.

Rely. The rely conditions can be automatically changed to enforce that the controlled arguments to do not change. In particular, if `R` is the programmer-written rely condition, then the SCOOP-specific rely condition is $R \wedge \forall c \in cs.$ `c.is_equal(old c)`, where `cs` are all the controlled objects. The advantage to this approach is that no modifications have to be done to the demonL tool. The disadvantage is that is somewhat susceptible to the implementation of `is_equal`,

```
wait_decr (c: separate COUNTER)
  require
    c.value > 1
  do
    c.decrement
  end

just_decr (c: separate COUNTER)
  do
    c.decrement
  end
```

(a) Waiting and immediate decrements to c

```
c: separate COUNTER

decr_twice
  do
    -- c.value is at least 1
    -- after this call
    wait_decr (c)

    -- However, another decrement
    -- may occur from another
    -- client before this call
    just_decr (c)
  end
```

(b) Client code that may fail

Figure 4.8: Atomicity violation in SCOOP

which may not always obey the Leibniz rule that $x = y \rightarrow f(x) = f(y)$; in other words: equal things cannot be distinguished. This property is important because demonL will just maintain that the `is_equal` notion of equality holds. If equal things can be distinguished, then demonL is free to change those parts that fall "outside" the `is_equal` definition. For example, if `is_equal` defined in a pair of numbers only compared the first component, then demonL would be free to change the second component.

demonL. If demonL is changed to accommodate this interpretation of the rely condition, then it would involve both changing the goal language to include a new field to specify unchanging objects, and the translation mechanism to transfer this knowledge to the SMT solver. The benefit of the approach is that there is a possibility for the SMT solver to encode these conditions more efficiently than if it has to rely on the specification of a `is_equal` query. However, it is a much more invasive change.

The approach that is taken is the first: use the rely condition to encode the controlled arguments in SCOOP. This makes for a more flexible process, as it reuses what is already there.

4.3.2 How SCOOP makes it easier

As mentioned previously, every controlled object effectively becomes "immune" to interference. This is fortunate, because all non-`separate` objects and all `separate` objects that are arguments to a routine or a `separate` block are controlled. Phrased another way, the only things that have to be mentioned in a rely condition are those that are `separate` and not visibly controlled.

For example, there are different ways to fix Figure 4.8. One way is to update the rely condition to reflect the global behaviour of the program. This approach is shown in Figure 4.9a. Such a fix is possible, although it may not be ideal if the rely-condition does not truly reflect the global behaviour of the program. However, in SCOOP, one can also use Figure 4.9b which brings the shared

```
c: separate COUNTER
decr_twice
  do
    -- c.value is at least 1
    -- after this call
    wait_decr (c)

    -- The rely condition
    -- prevents interference
    just_decr (c)
  ensure
    rely: c.value >= old c.value
  end
```

(a) Specification fix for `decr_twice`

```
decr_twice (c: separate COUNTER)
  require
    c.value > 1
  do
    -- c.value is at least 1
    -- after this call
    wait_decr (c)

    -- The guarantees still
    -- carry over here because
    -- ‘c’ is controlled
    just_decr (c)
  end
```

(b) Controlled fix for `decr_twice`

Figure 4.9: Fixing via specification and control SCOOP

`separate` object `c` under the control of the `decr_twice` routine. The second fix has the nice property that it no longer relies on the global behaviour of the program, it will always be a valid fix, because it changes the semantics of the program to exclude the atomicity violation. In a traditional threaded program this could be attained by the use of the mutex, although the mutex would have no direct connection to the data being manipulated. The mutex would have to be associated to the data by manually indicating this relationship in the rely condition.

4.3.3 Handling wait conditions

As with the case of regular threaded programs, there is an issue with testing that sometimes a routine must not only be resilient to the interference generated by other threads, but it may actually *require* those threads to do something to make progress. Usually this is expressed through a condition variable or other communication mechanism. In the threaded case the solution is to strive for minimal dependencies on other system components through the use of mock or stub objects.

The same approach should be used for SCOOP programs on uncontrolled wait-conditions. Since they are uncontrolled, the behaviour of the program should be to pause until they are satisfied. However, obviously in a setting where all objects are on the same handler, the precondition would just fail normally because the handler is controlled. This is why those preconditions should operate in a “stubbed” mode where instead of being an evaluation of program state, they drive the program state to being satisfied.

4.4 DemonL

Program synthesis constructs a program that satisfies a given specification. Demonic testing uses program synthesis to construct interference, where the pro-

```
type Idle_Counter {num_idlers: Integer}
increment (this: Idle_Counter)
 . . .
decrement (this: Idle_Counter)
 require
  non_zero: this.num_idlers > 0
 ensure
  this.num_idlers = old this.num_idlers  1
```

(a) Domain specification

```
this: Idle_Counter

initial
 not (this = null) and
 this.num_idlers = 1

final
 not (this.num_idlers > 0)
```

(b) Goal specification

Figure 4.10: The `IDLE_COUNTER` class in demonL

gram that is synthesized represents another thread that causes errors in the current thread.

4.4.1 The domain description language

Facilitating the demonic testing approach are a language and tool: *demonL*. In the same spirit as the verification language Boogie [5], demonL serves as an intermediate language to express the allowable types of interference. The input to the tool consists of two parts: a domain and a goal.

The domain language is as follows:

Domain	::=	$[\mathit{TypeDecl} \mid \mathit{ProcDecl}]^*$
TypeDecl	::=	**type** $\mathit{ident}\ \{\mathit{Decl}^*\}$
Decl	::=	$\mathit{ident} : \mathit{ident}$
ProcDecl	::=	$\mathit{ident}(\mathit{Decl}^*)[: \mathit{ident}]?\ \mathit{Pre}?\ \mathit{Post}?$
Pre	::=	**require** $\mathit{TaggedExpr}^*$
Post	::=	**ensure** $\mathit{TaggedExpr}^*$
TaggedExpr	::=	$\mathit{tag} : \mathit{Expr}$
Expr	::=	$\mathit{op}\ \mathit{Expr} \mid \mathit{Expr}\ \mathit{op}\ \mathit{Expr} \mid \mathit{Call}$
Call	::=	$\mathit{ident}(\mathit{Expr}^*)$

where *op* can be the common infix and prefix operators, with the addition of an `old` prefix operator.

The following goal language specifies the initial and final states that the planner must transition between. It shares the same expression and declaration syntax as the domain format.

Goal	::=	$\mathit{Decl}^*\ \mathit{InitialState}\ \mathit{FinalState}$
InitialState	::=	**initial** Expr^*
FinalState	::=	**final** Expr^*

Example 10 Figure 4.10 contains two demonL files that are required perform synthesis. Figure 4.10a shows the translation of the class `IDLE_COUNTER` (Figure 4.4) to the domain language. Figure 4.10b displays an associated goal specification, which uses the actions and definitions given in the domain.

The domain describes the state through data structures, functions on the state, as well as actions that transform the state. Actions and functions are described with pre- and postconditions. The goal declares the entities in the system and the constraints on the initial state and final state of those entities. The final state relates the initial and goal states through the use of `old` operator, which references the values in the initial state.

From a given pair of domain and goal file, demonL constructs an initial state that satisfies the initial constraints, a final state that satisfies the final constraints, and a series of actions whose pre- and postconditions are satisfied in the initial, final, and any intermediate states.

Example 11 To find the possible interference that could be used to destabilize Figure 4.4, the goal specification found in Figure 4.10 is used. The goal specification contains the negation of the precondition of the `decrement` operation, here: `this.num_idlers <= 0`. However, if the goal includes the rely-condition restricting the interference to only non-decreasing effects on the number of idle workers, then the program is correct under the rely assumption:

```
final
 this.num_idlers >= old this.num_idlers and
 not (this.num_idlers > 0)
```

This expression has the same structure as the stability criterion,

$$\exists \sigma'.\ R_\sigma(\sigma') \wedge \neg p(\sigma')$$

where

$$R_\sigma(\sigma') = \sigma'(\text{this}).\text{num\_idlers} \geq \sigma(\text{this}).\text{num\_idlers}$$

and

$$p(\sigma') = \sigma'(\text{this}).\text{num\_idlers} > 0.$$

4.4.2 Interference as satisfiability

The construction of the imperatively-styled planning tool, demonL, was undertaken to provide two features: handling of incomplete postconditions and to make a domain language available that is more suitable for modelling the data/routine structure of imperative programs. Current work loosening the requirements on action effects [88, 101, 25] does not offer the flexibility demanded by many programming specifications. In addition to more expressivity, the demonL language also allows C-style structures to be defined, to further offer a domain language that is suitable for more direct translation from imperative languages.

Originally, this work was based on a traditional planning tool for encoding the assertions from the programming language. However, this approach was not expressive enough to represent typical specifications. To address this, demonL encodes planning as a satisfiability problem [47] to be decided by an SMT-based reasoning tool.

The encoding of the demonL and goal as a satisfiability problem is given by the **compile** function. It works on a 5-tuple of program inputs: the domain types D, functions F, actions A, initial-state I, and goal-state G. Although they are presented together here, the initial- and goal-states reside in the goal file, and the types, functions, and actions are contained in the domain file.

$$\begin{array}{ll}\mathbf{compile}(D, F, A, I, G) = & \\ \quad \mathbf{types}(D) & \wedge \\ \quad \mathbf{tags}(A) & \wedge \\ \quad \mathbf{frame}(D) & \wedge \\ \quad \mathbf{function}(F) & \wedge \\ \quad \mathbf{actions}(A) & \wedge \\ \quad \mathbf{initial}(D, F, I) & \wedge \\ \quad \mathbf{goal}(D, F, G) & \end{array}$$

A common scheme in the formula is to range over a time index. This allows the actions to relate the pre- and post-state, and also to precisely specify the number of steps between the initial and final states.

Types The first aspect of the translation is to encode the information from the domain's types, capturing the fact that each value of a type is: distinct, typed, and has a notion of equality. To do this, each data-type from the domain is converted into an n-element enumerated type. For example, for a type T, the enumeration would have the distinct values $v_{1,T}, v_{2,T}, \ldots, v_{n,T}$, To enforce the distinct property, the auxiliary predicate **distinct** ensures that each value of a type is not equal to any other value.

$$\mathbf{distinctTypes}(D) \triangleq \mathbf{distinct}(\bigcup_{T \in D} \{v_{1,T}, v_{2,T}, \ldots, v_{n,T}\})$$

Additionally $\mathbf{isType}_T$ defines a family of type-check predicates: $\mathbf{isType}_T(v) \triangleq v_{1,T} = v \vee v_{2,T} = v \vee \ldots \vee v_{n,T} = v$. Lastly, each type gives an equality predicate, which compares two objects of a type, at two different time indices.

$$\mathbf{equalities}(D) \triangleq \bigwedge_{T \in D} eq_T = \lambda\ i\ j\ v_1\ v_2.\ \bigwedge_{f \in T_{fields}} T_f(i, v_1) = T_f(j, v_2)$$

The equality predicates generated are important to express what does and does not change as a result of a particular action. demonL models structures as a set of functions, *fields* that operate on types, each representing a field of the structure.

The following expression is the full the restrictions on types, namely that the values are distinct and the structural equality for all the data-types:

$$\mathbf{types}(D) \triangleq \mathbf{distinctTypes}(D) \wedge \mathbf{equalities}(D)$$

Action Tags The first step is to generate a tag for every action. This tag will be used to identify which action occurred at what time. The **tags** transformation, like the **types** one, generates unique values for action tags.

$$\mathbf{tags}(A) \triangleq \mathbf{distinct}(\{tag_a \mid a \in A\})$$

Framing As every action may have some effect, every action must stipulate what it changes and what it does not. This is accomplished through a time-indexed predicate that asserts which fields of objects remain the same. It takes a time index i and a predicate P. For every value v, if predicate $P(v)$ is true, then v's attributes are allowed to change. Otherwise, v must not change between time-steps i and $i+1$. As with the equalities, these frame predicates belong to a family, indexed by each type in the domain. All of the type-based frames are combined into a single frame condition that is used by actions

$$\begin{aligned}\mathbf{frame}(D) \triangleq \mathit{innerFrame} = \lambda\ P\ i.\ \forall\ v \\ \bigwedge_{T \in D} \mathbf{isType}_T(v) \land \neg P(v) \rightarrow eq_T(i, i+1, v, v)\end{aligned}$$

Actions Actions require some bookkeeping when they are translated, as they must encode the frame, tag assignment, and state modifications, via the postcondition.

$$\begin{aligned}\mathbf{actionSet}(A) \triangleq \bigwedge_{a \in A} a_{name} = \\ \lambda\ i\ a_{args}.\ \mathbf{actionFrame}(i, a) \land \mathit{actionAt}(i) = \mathit{tag}_a \land \\ \mathbf{expr}(a_{pre}, i, i) \land \mathbf{expr}(a_{post}, i, i+1)\end{aligned}$$

The **actionFrame** provides an action-specific frame that will be expressed in terms of the *innerFrame* predicate. The tag for the action is asserted to occur at time i through the *actionAt* function. Finally, the precondition is asserted to be true at i through **expr** and the postcondition relates the states at i and $i+1$. The arguments to the **expr** translation give the time indices to use for `old` and non-`old` expressions, respectively. The `old` modifier expresses that the expression that follows is evaluated in the pre-state of the action.

To handle action frames, the idea of weak-equality is required. Weak-equality is equality on objects, except for the fields in F,

$$\mathbf{weakEq}(e, F, v, i) \triangleq e = v \rightarrow \bigwedge_{f \in T_{fields} - F} T_f(i, e) = T_f(i+1, v)$$

Using this, a frame-predicate can be defined to be given to *innerFrame*. This frame-predicate collects all the relevant frames for the postcondition of the action, using the notion of weak-equality.

$$\mathbf{collectFrame}(i, a) \triangleq \lambda v\ .\ \bigwedge_{(e,F) \in \mathbf{access}_{expr}(a_{post})} \mathbf{weakEq}(e, F, v, i)$$

The $\mathbf{access}_{expr}$ function collects the expressions where the outermost syntax is a field access: those of the shape $x.f$. For example, $\mathbf{access}_{expr}(x.f + x.g = y.m) = \{(x, \{f, g\}), (y, \{m\})\}$. The **collectFrame** predicate describes when an object should remain unchanged by an action. It does this by asserting that if an object is the same as one of the ones in the action's postcondition, then only the fields that are mentioned in the postcondition can change.

Lastly, the collected frame-predicate is given with the time-index to the *innerFrame* predicate. This gives the frame for a particular action a and time-index i.

$$\mathbf{actionFrame}(i, a) \triangleq \mathit{innerFrame}(\mathbf{collectFrame}(i, a), i)$$

Functions The translation of domain functions is fairly straight-forward. Their translation is a simplified version of the one used for actions, owing to the fact that functions are pure in demonL and thus do not require extra framing predicates. Therefore, in this encoding functions are really just an expression involving the time-indexed attributes of the function's arguments. The post-conditions of functions must be of the form `Result = e`.

$$\mathbf{function}(F) \triangleq \bigwedge_{f \in F} f_{name} = \lambda i, \vec{as} f.\ \mathbf{expr}(f_{post}, i, i)$$

Finally, these pieces have to be combined to allow the SMT solver to build a path from the initial state to the goal state. This is done through an action predicate,

$$\mathbf{actions}(A) \triangleq action = \lambda\ i.\ \bigvee_{a \in A} a(i, args) \wedge \mathbf{actionSet}(A)$$

Each action describes the constraints placed on predicates at index i and $i+1$. These are then used in a contiguous conjunction of the form $\bigwedge_{i \in 0..n-1} action(i)$. Here, $args$ are free variables the SMT solver will instantiate as arguments to the action. This is the primary expression for the SMT solver to satisfy, ensuring that an action occurs at every step, and that a state can be constructed that satisfies the pre and postcondition of every action.

Initial and final state The initial state, which must be a list of Boolean expressions, is translated using $\mathbf{expr}(e, 0, 0)$ for each expression e. Similarly, the goal state is translated using the same technique, but the time indices are set to 0 and n. As with translating post-conditions, this allows goal states to refer to values at both the initial and final time indices, through the use of `old`.

As of yet, other fundamental planning optimizations such as graph analysis [9, 48] are not included, as the primary focus of the work was to provide a domain language that is closer to the language of contracts that is found in typical programs.

Example Figure 4.11 shows an example demonL domain for a simple counter.

```
type Counter {num: Integer}

is_empty (c: Counter): Boolean
  ensure Result = (c.num = 0)

decr (c: Counter)
  require c.num > 0
  ensure c.num = old c.num  1

x: Counter
initial x.num = 2
final x.num < 1
```

Figure 4.11: A simple counter

The counter that can be decremented and queried to see if it is empty (equal to zero).

The result of translating Figure 4.11 to SMT is given in Figure 4.12. The last

$$
\begin{aligned}
& v_{1,Counter} \neq v_{2,Counter} \wedge v_{2,Counter} \neq v_{3,Counter} \wedge v_{3,Counter} \neq v_{1,Counter} \\
\wedge\ & eq_{Counter} = \lambda\ i\ j\ v_1\ v_2.\ Counter_{num}(i, v_1) = Counter_{num}(j, v_2) \\
\wedge\ & tag_{incr} \neq tag_{decr} \\
\wedge\ & \left(\begin{aligned} & innerFrame = \lambda\ P\ i.\ \forall o.\ \neg P(o) \\ & \qquad\qquad \rightarrow \forall v.\ (v = v_{1,Counter} \vee v = v_{2,Counter} \vee v = v_{3,Counter}) \\ & \qquad\qquad \rightarrow eq_{Counter}(i, i+1, o, v) \end{aligned} \right) \\
\wedge\ & is\_empty = \lambda\ i\ c.\ Counter_{num}(i, c) = 0 \\
\wedge\ & \left(\begin{aligned} & decr = \lambda\ i\ c\ .\ innerFrame(\lambda\ v.\ c = v \rightarrow True, i) \\ & \qquad\qquad \wedge actionAt(i) = tag_{decr} \\ & \qquad\qquad \wedge Counter_{num}(i, c) > 0 \\ & \qquad\qquad \wedge Counter_{num}(i+1, c) = Counter_{num}(i, c) - 1 \end{aligned} \right) \\
\wedge\ & action = \lambda\ i.\ decr(i, decrArg1(i)) \\
\wedge\ & isType_T(x) \\
\wedge\ & Counter_{num}(0, x) = 2 \wedge Counter_{num}(2, x) < 1 \\
\wedge\ & action(0) \wedge action(1)
\end{aligned}
$$

Figure 4.12: Translation of counter example

three lines of the translation correspond to the type declarations, initial and final states. The remaining top-portion of the translation contains the data, functions, and action definitions from the domain.

4.4.3 The tool

The output of the tool is the sequence of actions, and their arguments, that bring the program from the initial to the final state. Given the specifications in Figure 4.10, this would be a call to `decrement`. If the underlying SMT solver reports that the constraints are unsatisfiable, this indicates that no sequence of actions could be found. To avoid long synthesis times, the tool constructs sequences bounded by number of instructions and number of unique references for each user-constructed type.

However, because of the constraint-based nature of the encoding, first the tool solves the interference problem in a single step with *no* actions to constrain the transformation. If it finds that the initial/final state constraints are unsatisfiable with no actions to further constrain them, then this is a proof that no interference can occur. If the constraints are satisfiable, then the tool tries to obtain the sequence of actions. This means that in demonL the number of actions in the interference do not affect the determination of if interference is impossible. If the tool says it is impossible, it is impossible for the given rely condition, and initial and final state. In short, there are essentially three answers that demonL can give:

no interference for the given rely condition and initial/final states.

interference is possible, and here are a sequence of actions which will cause it.

interference may be possible, but the actions which give rise to it could not be found. This is either because the bound on the sequence length of actions is too low or there are no actions in the system which can interfere.

Having an intermediate language and tool offers substantial advantages to the application of the demonic technique: separating the complexity of encoding the verification conditions from the task of routine instrumentation, and the possibility to target more than one source language and and more than a single solver in the back-end. The current technology choices for demonL are Eiffel as a source language to be translated to demonL, and Yices [30] as the SMT solver.

demonL is similar to planning tools. In particular it allows the movement from an initial state to a final state by a series of actions. However, the specification of the initial state and the actions are permitted to be weaker than generally allowed by planning tools that use languages such as the Planning Domain Definition Language (PDDL) [64]. Where PDDL only allows the effect of an action to be expressed using certain atomic-terms demonL has no such restriction: any expression can be used to describe the effect of an action. For example, where a PDDL domain would require a post-condition such as `attribute = 5`, demonL is able to deal with with post-conditions such as `attribute > 3`. demonL also does not assume determinism of the actions. These qualities are important when representing program specification, which are typically incomplete.

demonL is available for download from [102].

4.5 Experimental evaluation

It is essential for a testing technique to be judged by its reaction to bugs that occur in real software. For this purpose, a selection of bugs from a concurrency bug database [110, 23] are used to determine if demonic testing can detect and help form fixes for the faults. No particular criteria was used to select bugs from the database, besides striving for an overall diversity of faults. All experiments were carried out on an Intel Q6600 2.4GHz with 4GB of RAM.

4.5.1 Conversion from source programs

All of the test cases are extracted from real projects and translated into Eiffel. Since well-known concurrent applications with specifications are rare, the non-essential elements are removed from well-known code then it is converted to Eiffel and contracts are added. This is also done to enable the analysis of bugs from many languages, while minimizing the differences due to language features. To see an example of this process, the original Apache C-code for the running example is given in Figure 4.13. The main differences are the removal of the recycled pool functionality, and the removal of the explicit return-value checking of concurrency primitive (locks, condition variable) operations that is typically handled by exceptions in languages that support them.

4.5.2 Results

Table 4.2 lists the collection of concurrency bugs that are used to perform the evaluation; the first seven are from the bug database, with the last being a well-known Java standard library bug. All bugs have been replicated using the

```
apr_status_t ap_queue_info_wait_for_idler
  (fd_queue_info_t *queue_info,
   apr_pool_t **recycled_pool)
{
  apr_status_t rv;
  *recycled_pool = NULL;
  if (queue_info->idlers == 0) {
    rv = apr_thread_mutex_lock(queue_info->idlers_mutex);

    if (rv != APR_SUCCESS) {
      return rv;
    }

    if (queue_info->idlers == 0) {
      rv = apr_thread_cond_wait(queue_info->wait_for_idler,
                                queue_info->idlers_mutex);

      if (rv != APR_SUCCESS) {
        apr_status_t rv2;

        rv2 = apr_thread_mutex_unlock(queue_info->idlers_mutex);

        if (rv2 != APR_SUCCESS) {
          return rv2;
        }

        return rv;
      }
    }

    rv = apr_thread_mutex_unlock(queue_info->idlers_mutex);

    if (rv != APR_SUCCESS) {
      return rv;
    }
  }

  apr_atomic_dec32(&(queue_info->idlers));

  ... recycling of data structures
}
```

Figure 4.13: Original `wait_for_idle` routine from Apache

demonic testing technique, with the exception of MySQL #169, as explained in the discussion at the end of the section. Inspired by the AutoTest approach, work initially began using Eiffel as the source language; to broaden the scope of the evaluation bugs were translated from multiple other languages. These examples are available for download [102].

				Annotation			
	Program/Bug	Bug type	LOC	Lock	Simple	Complex	Time(s)
1	Apache #21285	Atomicity	125	0	4	0	0.982
2	Apache #25520	Data-race	101	0	2	0	0.124
3	Apache #45605	Data-race	227	1	4	0	0.217
4	MySQL #169	Atomicity	69	–	–	–	–
5	MySQL #644	Data-race	124	0	3	1	0.939
6	MySQL #791	Data-race	113	0	1	0	0.139
7	pBZip2	Order	168	1	1	0	2.289
8	Java Vector	Data-race	70	0	2	0	0.032

Table 4.2: Bug collection

The time taken to generate interference, or determine that none exists, was measured for the bugs that were successfully tested. The average time taken was 100ms for each request to demonL. This time is different from the times in Table 4.2, as each time in the table may include many requests to demonL.

4.5.3 Annotation complexity

In any approach which requires the addition of specification via program annotation, the burden that this annotation places on the programmer is a relevant consideration. Although difficult to measure objectively, the annotations are placed into three categories:

- Lock – a rely-annotation denoting that a lock protects some shared data from change by another thread.
- Simple – a non-concurrency-related program annotation stating a property of the program that is either a non-null check for a reference, or a linear equation.
- Complex – a non-linear expression, or a frame condition that is necessary to limit the scope of an operation.

Table 4.2 collects the types of annotations required in the test cases. These are the types of annotations required for demonic testing to give the correct cause of the bug in the full program in the cases of Apache, MySQL, and pBZip2. In the Java vector implementation one of many possible causes is given, as it is part of a library.

4.5.4 Discussion

The Apache bug #45605 example is notable due to the the double-check present in the `wait_for_idle` routine. Separate tools exist to classify some data-races as "potentially benign" [77]; the double-check pattern is benign and difficult for pure data-race checkers to deal with. Demonic testing does not require any secondary approaches to accomplish this: the determination of benign vs. malignant data-races is based on the program contracts.

The only bug from the test set which could not be discovered using demonic testing came from MySQL bug #169. The reason is that the invariant of the program could not be expressed without either ghost variables or artificially adding more data.

Incorrectly stated rely conditions will lead to both false-positives and false-negatives, as these are assumed to be true in the routine where they are defined. However, as in Table 4.2, all rely conditions required were quite simple, indicating that a certain lock protects some shared state.

The bounded synthesis done by demonL may affect the results by not considering interference from sequences of instructions that exceed the bound. However, this bound concerns the search for instruction sequences; there is also an initial unbounded-verification that demonL performs to determine the stability before trying to synthesize interference. The worst case is that the tool is unable to find the sequence of actions but still reports whether interference is possible. All bugs our evaluation examined required only interference of a single action to manifest themselves. This suggests that many concurrent bugs manifest themselves with little prompting; and that causing errors to present themselves in threaded executions is difficult due to scheduling rather than maintaining very complicated invariants.

The evaluation is limited by the number and selection of examples, it is possible that drawing on a larger pool of examples would offer greater insight into the properties of demonic testing. However, the small sample size is mitigated by the wide variety of types of concurrency bugs. Although every effort was made to make a faithful reproduction of the programs in the target language, there is the possibility transcription errors while moving between different programming languages.

4.6 Related Work

The idea of using routine specifications to discover concurrency errors is not unique to demonic testing. The Colt tool [95] for Java also uses this approach. However, their approach is less general, relying on hard-coded specification of the existing Java concurrent collection classes. Demonic testing is more generic as it works with user-supplied classes and specification, and as well allows finer-grained control of what constitutes an error through the usage of rely-conditions.

A common practice for testing of concurrent programs is load or stress testing. This frequently proves to be ineffective as in typical testing environments interleavings might only change marginally from one test run to the other. To force different interleavings, Edelstein et al. [31] present the ConTest tool, which combines a technique for deterministic replay of concurrent programs [19] with a heuristic for varying thread schedules by seeding sleep calls at synchronization points in the program.

Dynamic model checking [36, 74, 100] provides a more systematic approach by systematically exploring all possible thread interleavings. The search is stateless in that it provides a specialized scheduler that runs the program in its real execution environment, and hence can avoid storing concrete program states. The main problem is to overcome state explosion, which makes brute-force exhaustive search infeasible for large applications. Techniques such as partial order reduction as employed in the VeriSoft tool (Godefroid [36]) or preemption bounding (giving priority to schedules with fewer preemptions) in the CHESS tool (Musuvathi et al. [74]) can mitigate the effects of state space explosion only to a small degree. Wang et al. [100] propose a heuristic where ordering constraints learned from successful runs are used to guide the selection of interleavings for future runs. All of the above works focus on varying thread interleavings to produce undesired behaviour. Demonic testing differs from this approach by considering the routines in a program and finding sequences which lead to the violation of a program invariant, avoiding an exhaustive search of interleavings.

AutoBlackTest [62], while not directed at concurrent testing, is similar to the demonic testing in that sequences are constructed from a library of actions, although differs in that the actions are chosen to maximize general state-space exploration, and do not work towards finding any particular faults as demonic testing does.

A number of works use combinations of dynamic and symbolic analyses to improve testing of concurrent programs. Sen and Agha [94] use a combination of concrete and symbolic execution, termed concolic execution, to test multithreaded Java programs with the tool jCUTE. Symbolic execution produces input values that guide the concrete execution to alternate paths; concrete execution guides the symbolic computation along a concrete path to concretize any values that cannot be handled by a constraint solver. Besides producing alternate input values, their technique also systematically generates thread schedule variations such that potentially all causal structures of a concurrent program can be explored. Sen [93] introduces RaceFuzzer, an algorithm which uses race warnings from race detection tools to create problematic interleavings during testing in order to eliminate false positives automatically. Park et al. [80] propose CTrigger, a testing tool to expose atomicity violation bugs. The tool analyzes traces to find unserializable interleavings then these interleavings are explored during testing to expose bugs. Kundu, Ganai, and Wang [54] present a framework that combines conventional testing with symbolic analysis. A test harness invokes the program with random test values. Concrete traces are relaxed into concurrent trace programs, which capture all linearizations of events that respect the control flow of the program. The concurrent trace programs are then symbolically verified. All these techniques combine in some way the dynamic execution of programs with symbolic computation and verification, and most closely resemble the work presented in this work. However, they, like all other related work shown, are not able to achieve truly modular testing of concurrent software; they all depend on multithreaded executions or traces.

Contracts have been used successfully in unit testing of sequential software [68], where they can provide test oracles and filter inputs for random testing. Araujo et al. [3] evaluate the use of contracts in a concurrent setting, based on an extension of the JML [56] contract semantics. They found contracts as test oracles effective in finding and diagnosing concurrency-related faults on an industrial case study in Java/JML. In contrast to this work, demonic testing emphasizes the use of contracts also for symbolic analyses, in addition to test oracles.

Rely-guarantee reasoning has been applied in testing of concurrent programs.

Dingel [29] uses the state exploration tool VeriSoft [36] for rely-guarantee verification of C/C++ components. The component code is executed in parallel with an environment which generates initial states, monitors the component execution, and generates responses. If a program step is found to violate one of the guarantees, a flaw is found. Blundell et al. [12] use labelled transition systems to model the behaviour of components, whereas demonic testing works directly on source code. Assumptions on the model-level are used as environments in which individual components are executed. The execution results in traces which are in turn checked against the guarantees of the model. Failure of a check suggests an incompatibility between a model and its implementation.

4.7 Conclusion

The testing of concurrent systems has generally been regarded as inferior to static approaches as a method of increasing the level of correctness. The realization that purely static reasoning also faces problems of scalability or precision when applied to concurrent systems has led to a more pragmatic assessment, leaving testing its due place, as evidenced by the approaches reviewed in the previous section.

Unlike many of these approaches, which are only suitable for testing entire systems, demonic testing addresses the important problem of unit testing for concurrent programs. Through its combination of dynamic and symbolic techniques, demonic testing provides two significant benefits over other proposals. First, it makes use of the available testing tools and existing tests for sequential programs. Second, instead of searching the state space of thread interleavings, demonic testing uses program synthesis as a constructive means to find problematic thread interference. If a test fails, a test case and a problematic sequence of interactions is available for analysis.

The availability of contracts in Eiffel makes this a natural setting in which to provide demonic testing. The data race freedom of SCOOP also forms a valuable base upon which to build the technique, as it shows that it is not only valuable as a race detection technique but, in fact, also is suitable to detect atomicity violations in cases where races either do not, or can not, exist.

This work has been published, in part, at ICFEM 2012 [106].

Part II: Execution

Chapter 5

Efficient SCOOP

Programming languages and libraries that help programmers write concurrent programs are the subject of intensive research. Increasingly, special attention is paid to developing approaches that provide certain execution guarantees; they support the programmer in avoiding delicate concurrency errors such as data races or deadlocks. For example, languages such as Erlang [4] and others based on the Actor model [40] avoid data races by a pure message-passing approach; languages such as Haskell [82] are based on Software Transactional Memory [96], avoiding some of the pitfalls associated with traditional locks.

Providing these guarantees can, however, be at odds with attaining good performance. Pure message-passing approaches face the difficulty of how to transfer data efficiently between actors; and optimistic approaches to shared memory access, such as transactional memory, have to deal with recording, committing, and rolling back changes to memory. For this reason, execution strategies have to be developed that preserve the performance of the language while maintaining the strong execution guarantees of the model.

SCOOP places restrictions on the way concurrent programs execute, thereby gaining more reasoning capabilities but also introducing potential performance bottlenecks. To improve the performance of SCOOP programs while maintaining the core of the execution guarantees, this work introduces a new execution model called SCOOP/Qs[1]. First, a formulation of the SCOOP semantics is given. One which admits more concurrent behaviour than the existing formalizations [70], while still providing the reasoning guarantees. On this basis, lower-level implementation techniques are developed to make the scheduling and interactions between threads efficient. These techniques are applied both in an advanced prototype implementation [103], as well as incorporated into the research branch of the production EiffelStudio compiler [17], the reference compiler for SCOOP programs.

Section 5.1 examines the existing SCOOP model in terms of the high level guarantees that it provides, and how much concurrency a naïve model provides. Section 5.2 proposes a new semantics for handler reservation that enables greater amounts of concurrency, shows it as a member of a larger family of SCOOP semantics, and shows how deadlock behaviour changes between different members of the family of semantics. Section 5.3 shows how to compile efficiently using the SCOOP/Qs model. Section 5.4 explains the internal design of the concurrency run-time library for SCOOP/Qs. These techniques are applied both in an advanced prototype implementation [103], as well as incorporated into the research branch of the production EiffelStudio compiler [17], the reference compiler for SCOOP programs. The differences in the independent SCOOP/Qs run-time and the one built into EiffelStudio are also described. Section 5.5

[1]Qs is pronounced “queues”, as queues feature prominently in the new approach; the run-time and compiler associated with Qs is called Quicksilver, available from [103].

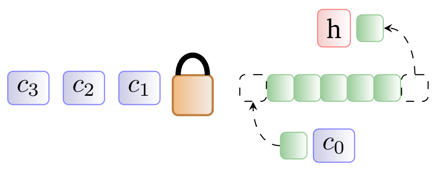


Figure 5.1: Normal handler implementation

discusses related work, and finally Section 5.6 concludes.

5.1 The SCOOP execution model

The key motivation behind SCOOP [78] is providing a concurrent programming model that allows the same kinds of reasoning techniques that sequential models enjoy. In particular, SCOOP aims to provide areas of code where pre/postcondition reasoning exists between independent threads. To do this, SCOOP allows one to mark sections of code where, although threads are operating concurrently, data races are excluded entirely.

5.1.1 Reasoning guarantees

There are a few key reasoning guarantees that an implementation of SCOOP must provide:

1. Regular calls (non-`separate`) calls and primitive instructions (assignment, etc.) execute immediately and are synchronous.
2. Calls to another handler, `x`, within the body of a `separate x` block will be executed in the order they are logged, and there will be no intervening calls logged from other clients.
3. The postcondition of calls to a set of non-`Current` handlers `xs`, arising from the calls logged within the body of a `separate xs` block can be assumed together if the next client to lock any `x` $\in$ `xs` locks all of `xs`.

The effect of rule 1 is that normal sequential reasoning is applied to calls that are issued by the client, to the client. Rule 2 implies that calls that are made from the client to the handler are applied in order, thus the client can and must apply pre-/postcondition reasoning from one call it has made to the next. Rule 3 states that to both give and receive reasoning guarantees that relate multiple handlers the handlers must be reserved together.

5.1.2 A naïve implementation

The original SCOOP operational semantics [78] mandated the use of a lock to ensure that pre/postcondition reasoning could be applied by a client on its calls to a handler. One can visualize this as the client c_0 placing the calls in a queue for the handler h to dequeue and process, as in Figure 5.1. The other clients (c_1, c_2, c_3) that may want to access the handler's queue must wait until the current client is finished.

The lock-based model of SCOOP satisfies the reasoning guarantees by using locks to make sure that there is only ever one client that can add calls to a handler's work queue. The guarantees for multiple handlers can be reserved simply by taking the locks in a predefined order.

5.1.3 Issues with blocking

SCOOP, when implemented using locks, has the disadvantage that the basic operation required to make use of concurrent execution, reserving a handler, requires a blocking operation. This has two major effects: it is easier to run into deadlock because locks are *everywhere* and, because locks ensure mutual exclusion, it puts more clients to sleep while they wait to acquire the handler, even if they only want to send the handler requests, not receive any answers.

5.2 A model with less locking

The first SCOOP guarantee is easy to achieve, it is simply how sequential programs operate. To understand how to implement SCOOP efficiently, it is important to concentrate on the second guarantee. This guarantee states that requests from a particular client are processed in the order they are sent, disallowing interleaving requests from other clients. Non-interference of different clients can be achieved by giving each client their own private area (a queue) in which to place their requests. Each client then just shares this private queue with the handler to which it wants to send requests.

Syntax. The following syntax of expressions e is used to describe the execution model.

$$\begin{aligned} e ::= \;& \texttt{separate}\; x\; e\; e \mid \texttt{call}(x, f) \mid \texttt{query}(x, f) \mid \\ & \textsf{yield}\; h \mid \textsf{wait}\; h \mid \textsf{release}\; h\; e \mid \textsf{end} \mid \textsf{skip} \mid \textsf{if}\; e\; e\; e \end{aligned}$$

Note that `separate` blocks and `call` and `query` requests model instructions of SCOOP programs, whereas the expressions wait, release, end, yield, skip, and, if are only used to model the runtime behaviour. In particular, expressions wait and release describe the synchronization to wait for the result after a `query`, expression end models the end of a group of requests, yield models wait-condition behaviour, and skip models no behaviour.

Operational semantics. Figure 5.2 and Figure 5.3 specify an operational semantics[2] that conforms to the SCOOP guarantees. Figure 5.2 describes the rules that are specific to the SCOOP model, and Figure 5.3 describe those rules which are more general (sequencing, parallel composition, if). It is described using inference rules for transitions of the form $P \Rightarrow Q$, where P and Q are parallel compositions of handlers. The $||$ operator is commutative and associative to facilitate appropriate reordering of handlers.

The basic representation of a handler is a triple (h, q_h, e) of its identity h, request queue q_h, and the current program it is executing, s. A request queue is a list of handler-tagged private queues, and is thus really a queue-of-queues.

[2] The operational semantics shown in Chapter 3 examines the locking behaviour of SCOOP programs, where the one presently being described focuses on the way calls are transferred from clients to handlers.

$$\textsc{Separate}\ \frac{\begin{array}{c}\text{proceed} = e;\ \texttt{call}(x, \mathsf{end}) \\ \text{retry} = \texttt{call}(x, \mathsf{end});\ \mathsf{yield}\ x;\ \texttt{separate}\ x\ w\ e\end{array}}{\begin{array}{l}(h, q_h, \texttt{separate}\ x\ w\ e) \quad \|\ (x, q_x, t) \Rightarrow \\ (h, q_h, \mathsf{if}\ e\ \text{proceed retry}) \ \|\ (x, q_x + [h \mapsto []]\,, t)\end{array}}$$

$$\textsc{Yield}\ \frac{y \neq h}{\begin{array}{l}(h, q_h, \mathsf{yield}\ x) \ \|\ (x, [y \mapsto ys] + zs, t) \Rightarrow \\ (h, q_h, \mathsf{skip}) \quad\ \|\ (x, [y \mapsto ys] + zs, t)\end{array}}$$

$$\textsc{Call}\ \frac{}{\begin{array}{l}(h, q_h, \texttt{call}(x, f)) \ \|\ (x, q_x, t) \Rightarrow \\ (h, q_h, \mathsf{skip}) \qquad\quad \|\ (x, q_x\,[h \mapsto q_x[h] + [f]]\,, t)\end{array}}$$

$$\textsc{Query}\ \frac{}{\begin{array}{l}(h, q_h, \texttt{query}(x, f)) \ \|\ (x, q_x, t) \Rightarrow \\ (h, q_h, \mathsf{wait}\ x) \qquad\ \|\ (x, q_x\,[h \mapsto q_x[h] + [\mathsf{release}\ h\ f]]\,, t)\end{array}}$$

$$\textsc{ReleaseStep}\ \frac{(h, xs, e) \Rightarrow (h, xs, e')}{(h, xs, \mathsf{release}\ h\ e) \Rightarrow (h, xs, \mathsf{release}\ h\ e')}$$

$$\textsc{Sync}\ \frac{}{\begin{array}{l}(h, q_h, \mathsf{wait}\ x) \ \|\ (x, q_x, \mathsf{release}\ h\ \lfloor v \rfloor) \Rightarrow \\ (h, q_h, \lfloor v \rfloor) \qquad \|\ (x, q_x, \mathsf{skip})\end{array}}$$

$$\textsc{Run}\ \frac{}{\begin{array}{l}(h, [x \mapsto [e] + es] + ys, \mathsf{skip}) \Rightarrow \\ (h, [x \mapsto ss] + ys, e)\end{array}} \qquad \textsc{End}\ \frac{}{\begin{array}{l}(h, [x \mapsto []] + ys, \mathsf{end}) \Rightarrow \\ (h, ys, \mathsf{skip})\end{array}}$$

Figure 5.2: Inference rules of SCOOP/Qs

Private queues of a client handler c can be looked-up $q_h[c]$, and can be updated $q_h[c \mapsto l]$, where l is the new list to associate with the handler h. Both lookup and updating work on the *last* occurrence of c, which is important as this is the one that the client modifies. The queue can also be decomposed structurally, with $[x \mapsto s] + ys$ meaning that the head of the queue is from client x with private queue s, and ys is the rest of the structure (possibly empty). So although the private queues in the queue-of-queues can be accessed and modified in any order, they are inserted and removed in first-in-first-out order.

Figure 5.2 shows `separate` blocks (the rule SEPARATE), the two different kinds of requests (CALL, QUERY, SYNC rules), how a client retries a wait condition (YIELD), and how the handlers process these requests (RUN, REQUESTSTEP, and END rules). The rules from Figure 5.3 are defined in the standard way.

In the rule SEPARATE, clients insert their private queue at the end of the handler's request queue. This operation occurs at the beginning of a `separate` block. This registers them with the handler, who will eventually process the requests in it. The fact that a handler only processes one private queue at a time ensures that the reasoning guarantees are maintained. If the wait-condition is trivial (e is the constant True), then the body of the `separate` block is entered without waiting. Since supplier's handler-triple consist only of variables, and

$$\textsc{Seq}\ \frac{(h, xs, e_1) \Rightarrow (h, xs, e_1')}{(h, xs, e_1\,; e_2) \Rightarrow (h, xs, e_1'\,; e_2)} \qquad \textsc{SeqSkip}\ \frac{}{(h, xs, \mathsf{skip}; e_2) \Rightarrow (h, xs, e_2)}$$

$$\textsc{IfStep}\ \frac{(h, xs, c) \Rightarrow (h, xs, c')}{(h, xs, \mathsf{if}\ c\ e_1\ e_2) \Rightarrow (h, xs, \mathsf{if}\ c'\ e_1\ e_2)}$$

$$\textsc{IfTrue}\ \frac{}{\begin{array}{l}(h, xs, \mathsf{if}\ \lfloor \mathrm{True} \rfloor\ e_1\ e_2) \Rightarrow \\ (h, xs, e_1)\end{array}} \qquad \textsc{IfFalse}\ \frac{}{\begin{array}{l}(h, xs, \mathsf{if}\ \lfloor \mathrm{False} \rfloor\ e_1\ e_2) \Rightarrow \\ (h, xs, e_2)\end{array}}$$

$$\textsc{ParStep}_1\ \frac{P \Rightarrow P'}{P \mid\mid Q \Rightarrow P' \mid\mid Q} \qquad \textsc{ParStep}_2\ \frac{Q \Rightarrow Q'}{P \mid\mid Q \Rightarrow P \mid\mid Q'}$$

$$\textsc{OneStep}\ \frac{P \Rightarrow Q}{P \Rightarrow^* Q} \qquad \textsc{ManyStep}\ \frac{P \Rightarrow^* P' \qquad P' \Rightarrow^* Q}{P \Rightarrow^* Q}$$

Figure 5.3: Standard rules

variables will match anything, there are no restrictions on what state the supplier has to be in for this rule to apply. Also, the client appends a `call`$(x,$ `end`$)$ action after the `separate` block body to signal that the supplier x can take requests from other clients. When e is non-trivial, it is evaluated until it is a value and the program either re-evaluates the `separate` construct (again, possibly waiting), or enters the body of the block.

Executing wait conditions is something that can be done naïvely: repeatedly reserve, reevaluate the condition, and give up the reservation on the handler. Unfortunately in this simple approach, a client can be placed next to itself in the handler's queue-of-queues. This is unnecessary because the queries are assumed to be pure, and thus if the condition was not true the first time the client evaluated it, it will not be true now. This can be avoided if the semantics can be constructed to disallow placing a handler that has just finished a wait condition next to itself in the handler's queue-of-queues. This is what occurs in the YIELD rule.

The SCOOP/Qs semantics, in contrast to the original lock-based SCOOP semantics, uses multiple queues that can all be accessed and enqueued into simultaneously by clients. Figure 5.4 visualizes this behaviour. In the figure, the

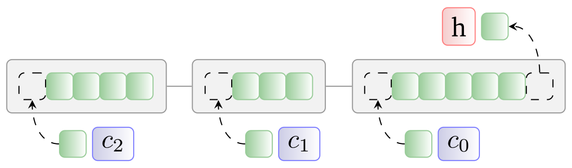


Figure 5.4: Handler implementation based on queue of queues

outer gray boxes are nodes in the queue-of-queues, and the inner green boxes are nodes in the private queues. This nested queueing maintains the reasoning guarantees while still allowing all clients to enqueue asynchronous calls without waiting.

In rule CALL, the `call` action is non-blocking: it asynchronously appends the requested method f to the end of the appropriate client's private queue.

Rule QUERY, requesting a query execution, however, does require blocking as it must wait for the result of the function application. This is modelled by sending the query request and introducing a pair of actions which can only step forward together: the wait/release pair. There is only one rule (SYNC) that can rewrite sync into a value, $\lfloor v \rfloor$, and release into a skip expression. This rule only applies when the client is currently evaluating sync and the handler is evaluating release with the second argument evaluated to a value. The RELEASESTEP rule evaluates the second argument of a release expression until it is able to be sent back to the client.

Each handler processes its request queue in the following way: in rule RUN, if the handler is idle (executing skip) then it will examine the request queue. If the request queue's first entry (a private queue) is non-empty, then the first action is taken out of that private queue and placed in the program part of the handler to execute. If the request queue is empty, or it contains an empty private queue as its first entry, then the thread does nothing. In rule END, the thread finishes one private queue and switches to the next when it encounters the end expression, which was placed by the owner of the outgoing private queue when it finished executing its `separate` block (rule SEPARATE).

For simplicity, the semantics in Figure 5.2 do not support reserving more than one handler per `separate` block. This is not an inherent limitation, the SEPARATE rule would become two rules, a rule that reserves the each handler in a list, and a base case that detects when all handlers are reserved and moves to processing the wait-condition. The YIELD rule and yield expression also expand to include a set of handlers to wait on, rather than just a single handler.

5.2.1 A family of models

The SCOOP/Qs operation semantics does not also describe the exact behaviour of the existing SCOOP semantics. The most obvious of which is the non-blocking reservation of handlers in SCOOP/Qs, because it only amounts to enqueueing a private queue in the queue-of-queues. To describe both models, the operational rules presented in Figure 5.2 have to be modified.

To use a single operational semantics to describe both the existing and the SCOOP/Qs behaviour, the queue-of-queues and the private queues must be able to set bounds on how many items they can each hold. The exact modifications to the rules can be seen in Figure 5.5. The additional parameters to the rules are M and N, representing the upper limit on the number of elements (this could also be interpreted as the total size of all elements, if measured in bytes instead of elements) allowed the private queues and queue-of-queues, respectively. The existing SCOOP implementation would be a member of this semantic family with $M = \infty$ and $N = 1$, representing that a handler will only ever have the private queue of an single client in its queue-of-queues. This also corresponds to the semantics in Chapter 3, and the results from there still apply to the $M = \infty$ and $N = 1$ family member. More permissive implementations may try to set $M = \infty$ and $N = \infty$, which are the parameters used in SCOOP/Qs.

These bounds are not only a theoretical concern, they also reflect the realities of implementing a distributed computation model like SCOOP. In particular, be-

$$\text{SEPARATE}\ \frac{\begin{array}{c}\text{proceed} = e;\ \texttt{call}(x, \mathsf{end})\\ \text{retry} = \texttt{call}(x, \mathsf{end});\ \mathsf{yield}\ x;\ \texttt{separate}\ x\ w\ e\\ |q_x| < N\end{array}}{\begin{array}{ll}(h, q_h, \texttt{separate}\ x\ w\ e) & || \ (x, q_x, t) \Rightarrow\\ (h, q_h, \mathsf{if}\ e\ \text{proceed retry}) & || \ (x, q_x + [h \mapsto []], t)\end{array}}$$

$$\text{CALL'}\ \frac{|q_x[h]| < M}{\begin{array}{ll}(h, q_h, \texttt{call}(x, f)) & || \ (x, q_x, t) \Rightarrow\\ (h, q_h, \mathsf{skip}) & || \ (x, q_x\,[h \mapsto q_x[h] + [f]], t)\end{array}}$$

$$\text{QUERY'}\ \frac{|q_x[h]| < M}{\begin{array}{ll}(h, q_h, \texttt{query}(x, f)) & || \ (x, q_x, t) \Rightarrow\\ (h, q_h, \mathsf{wait}\ x) & || \ (x, q_x\,[h \mapsto q_x[h] + [\mathsf{release}\ h\ f]], t)\end{array}}$$

Figure 5.5: Generalized SCOOP/Qs rules

tween a client sending a request and a handler dequeueing and processing it, the request must be stored. The storage is not free, and in some circumstances there may be limitations imposed by external sources, i.e., the outgoing packet buffer of the kernel's network queue. The current SCOOP/Qs run-time implementation currently uses $M, N = \infty$.

5.2.2 Deadlock behaviour in different family members

The SCOOP/Qs semantics ($M = \infty$ and $N = \infty$) represents a change in program execution that can influence correctness. In particular, since the original semantics ($M = \infty, N = 1$) of handler acquisition was blocking, programs could deadlock merely trying to acquire handlers. For example, the program in Figure 5.6 will deadlock under some schedules when using the original seman-

```
separate x
  do
    separate y
      do
        x.foo()
        y.bar()
      end
  end
```
Client 1

```
separate y
  do
    separate x
      do
        x.foo()
        y.bar()
      end
  end
```
Client 2

Figure 5.6: Possible deadlock situation

tics. This is due to the inconsistent locking order[3] of x and y. However, in the SCOOP/Qs execution model this example cannot deadlock because there are no longer any blocking operations: both clients can simultaneously reserve the handlers x and y, and log asynchronous calls on them.

The same reduction in deadlocks is not present with the inclusion of queries, though. The presence of queries introduces a dependency between the handler and client where the client must be at the top of the handler's queue-of-queues

[3] Preventing this manner of deadlock is discussed in Chapter 3.

for the query to return (sync finishes). This introduces the possibility for deadlock again. Figure 5.7 shows a program, in the same style as Figure 5.6, that may deadlock. This program will deadlock in both the $M = \infty$, $N = 1$ model as

```
separate x                  separate y
  do                          do
    x.query()                   y.query()
    separate y                  separate x
      do                          do
        y.query()                   x.query()
        x.foo()                     x.foo()
        y.bar()                     y.bar()
      end                         end
  end                         end
     Client 1                    Client 2
```

Figure 5.7: New deadlock situation

well as the $M = \infty$, $N = \infty$ model because in both cases the clients must wait until their private queues calls are first x's (resp. y's) queue-of-queues before proceeding past the `x.query` expression. Since the ordering of these waits are not between the two clients, there is the possibility for deadlock.

The SCOOP/Qs model does not even represent a strict reduction in deadlocking executions. There are programs where the $M = \infty$, $N = 1$ model will not deadlock, but the $M = \infty$, $N = \infty$ model will deadlock. Figure 5.8 shows such a program. This program deadlocks in the $N = \infty$ case because any number

```
separate z                      separate z
  do                              do
    separate x                      separate y
      do                              do
        x.query()                       y.query()
        separate y                      separate x
          do                              do
            y.query()                       x.query()
            x.foo()                         x.foo()
            y.bar()                         y.bar()
          end                             end
      end                             end
  end                             end
     Client 1                        Client 2
```

Figure 5.8: Deadlocking program in $N = \infty$ but not in $N = 1$

of clients may simultaneously proceed into the `separate z` block, which then engages in the classic deadlock situation shown in Figure 5.7. This behaviour even occurs when $N = 2$. However, in the case where $N = 1$, only one client can be in the body of the `separate z` at a time. Whichever client is in the body therefore has no dependency problems acquiring and having its query executed by the handlers of x and y. Therefore, there can be no deadlock in Figure 5.8 when $N = 1$.

Table 5.1 highlights the major design choices that can be made when selecting

M	N	Behaviour
∞	∞	Clients are assured that locking a handler and logging asynchronous calls will not ever block.
C	∞	Client can only have a C async call active at a time, but will always be able to lock a supplier without blocking. However, the client can reason about how many calls they log, so can control C locally.
∞	C	Client can log all calls asynchronously, but can only C clients can log calls on a handler at a time. This means that global information must be used to determine if the system will block and where.
C_1	C_2	The client can only log C_1 calls, and only C_2 clients can lock a handler simultaneously. This combines the properties of the above two cases.

Table 5.1: SCOOP-semantics family members

a member from the family of semantics. The basic choice is between allowing an unbounded queue length or to set some limit, $C < \infty$.

SCOOP/Qs makes it possible to construct fully deadlock free programs by abiding by a very simple rule, one that could even be checked automatically:

Only make asynchronous calls.

This is entirely feasible considering that all Erlang programs are written in this way, as Erlang has no synchronous messaging mechanism (that is not to say that Erlang programs guarantee liveness).

Previously, even programs that made only asynchronous calls could still deadlock due to the blocking arising from handler reservation. The introduction of non-blocking handler reservations makes it possible for programs that only use asynchronous calls to be guaranteed that they are completely free of deadlock.

5.2.3 Multiple handler reservations

The `separate` block as shown so far only reserved a single handler. It is also possible to reserve multiple handlers at the same time to ensure that the objects from those handlers stay related in the appropriate way.

Inspect, for example, the programs in Figure 5.9. One might expect that

```
separate x
  do
    separate y
      do
        x.setColour (Red)
        y.setColour (Red)
      end
  end
```

```
separate x
  do
    separate y
      do
        x.setColour (Blue)
        y.setColour (Blue)
      end
  end
```

Figure 5.9: Nested reservations

in such a program that no matter which thread goes first, that there would be an invariant that `x.colour = y.colour` whenever a thread has both `x` and `y`'s handlers reserved. This invariant is true in a locking model of handler reservation because while holding the lock on `x`'s queue the other client cannot attain a lock on `y`'s queue for the duration that the lock on `x` is held. However, this is not true with the queue-of-queues model because the nesting of reservations does not imply blocking. Thus, there is no guarantee that while thread 1 has attained its reservation in `x`'s queue-of-queues, that thread 2 does not grab a place on `y`'s queue-of-queues before thread 1 does. Thus, `x` may have $[[setColour(Red)], [setColour(Blue)]]$ as its queue-of-queues and `y` may have $[[setColour(Blue)], [setColour(Red)]]$. This means that after both handlers have finished processing all the calls `x.colour = Blue and y.colour = Red`.

However, there is a method of attaining the desired property in both SCOOP and SCOOP/Qs. That is to simply use the multiple reservation `separate` blocks. This is the recommended way in SCOOP of reserving multiple handlers as it eliminates the danger of deadlock due to reserving handlers in an order that would lead to mutual dependencies on the locks. The same ordering guarantees that maintained the desired cross-handler invariant previously is the also the same that leads to deadlock. Figure 5.10 shows how to rewrite the example in Figure 5.9 to use multiple reservation `separate` blocks. Written in this way,

```
separate x y                      separate x y
  do                                do
    x.setColour (Red)                 x.setColour (Blue)
    y.setColour (Red)                 y.setColour (Blue)
  end                               end
```

Figure 5.10: Multiple reservations

when executed under either SCOOP or SCOOP/Qs the program will neither deadlock nor exhibit an execution where `x.colour /= y.colour`.

The modification to the SEPARATE rule to support this is straight-forward. First one defines an update function that updates a handler if it is in the set X.

$$\text{resOne}(X, h, (x, q_x, t)) = \begin{cases} (x, q_x + [h \mapsto []], t) & \text{if } x \in X \\ (x, q_x, t) & \text{if } x \notin X \end{cases}$$

Then this is applied over the parallel composition of all handlers.

$$\begin{aligned} \text{resMany}(X, h, P \mid\mid Q) &= \text{resMany}(X, h, P) \mid\mid \text{resMany}(X, h, Q) \\ \text{resMany}(X, h, (x, q_x, t)) &= \text{resOne}(X, h, (x, q_x, t)) \end{aligned}$$

Lastly, a function describes that each handler in the set (represented here by a list so it can be traversed) is sent an end message.

$$\begin{aligned} \text{endMany}(x :: xs) &= \texttt{call}(x, \mathsf{end}); \text{endMany}(xs) \\ \text{endMany}([]) &= \mathsf{skip} \end{aligned}$$

These functions combine to define a generalized SEPARATE rule that can reserve multiple handlers atomically.

$$\textsc{separate}\;\frac{\begin{array}{c} P' = \text{resMany}(X, h, P) \\ \text{ends} = \text{endMany}(X) \end{array}}{\begin{array}{ll} (h, q_h, \texttt{separate}\ X\ s) & \mid\mid P \Rightarrow \\ (h, q_h, s; \text{ends}) & \mid\mid P' \end{array}}$$

5.3 Compiling SCOOP/Qs programs

The semantics described in Section 5.2 are used to implement a compiler and run-time for SCOOP programs. The operational semantics gives rise to notable run-time performance and implementation properties. SCOOP/Qs pays particular attention in moving the implementation from a synchronization-heavy model to one which reduces the amount of blocking.

The implementation of SCOOP/Qs follows the semantic description in Section 5.2 as closely as possible, including the small operational simplifications. The run-time is written in C, the compiler is written in Haskell and targets the LLVM framework [58] to take advantage of the lower level optimizations that are available. Using LLVM is a necessary choice for this work because it is important to compare with other, quite mature, languages and the comparison shouldn't focus on obvious shortcomings. It is also built for extension by means of adding optimization passes, which this work takes advantage of. The SCOOP/Qs compiler, run-time, and benchmarks are available from GitHub [103].

The run-time is broken into 3 layers: task switching, lightweight threads, and handlers. Some of the optimizations described in this section take place at the handler layer, but there are also some that use the other two layers as well to optimize scheduling.

5.3.1 SCOOP/Qs calling convention

There are two distinct pieces to a a SCOOP/Qs separate object, one is the actual object, which represents the program data, and the other is the object's handler, the entity which has to be asked for permission to access the object. To efficiently make separate calls, finding the handler for an object must be accessible very quickly. To achieve this, SCOOP/Qs has a convention that when a routine is compiled, the target of the routine will a pair of the target object and its handler. This is similar to how the implicit first argument of object-oriented routines is the `Current` or `this` object. Non-`Current` separate objects follow a similar pattern, they are represented as a tuple of object and handler reference.

Non-separate objects are not represented as a pair, they are represented just as they normally would be. This means that the `Current` handler must be carefully passed around when calls are made so that if a non-separate object wants to make a call to a separate-object, the appropriate handler to make the request from is always available.

5.3.2 Request processing

The RUN and END rules describe all of the queue management facilities that a handler has to perform. This correspondence is shown in the high-level implementation of the main handler-loop given in Figure 5.11.

The structure of the handler's loop directly corresponds to the data structure implementation (a queue of queues). One can see that private queues are continually taken from the outer queue, where the dequeue operation returns a Boolean result. False corresponds to no more work (indicating the handler can shut down), not that the queue is empty as may be in a non-blocking queue implementation. For each private queue that is received, calls are repeatedly dequeued out of it and executed until false is returned from the dequeue operation,

```
// RUN rule, when there is a private queue
//   available
while (qoq.dequeue (&private_queue))
  {
    // if dequeue returns true:
    //    RUN rule; process calls from
    //    this queue.
    // otherwise:
    //    END rule; switch to the next
    //    private queue
    while (private_queue.dequeue (&call))
      {
         execute_call (call);
      }
  }
```

Figure 5.11: Main handler-loop

indicating that the END rule has been triggered, and the client presently does not wish to log more requests.

Note that the arrangement of clients and handlers follows a particular pattern when the queue-of-queues pattern is used. Namely, that each handler first reserves a position in the queue-of-queues: each queue-of-queues has many clients trying to gain access, but only one handler removing the private queues. This is a typical multiple-producer single-consumer arrangement, so an efficient lock-free queue specialized for this case can be used to implement the queue-of-queues. Similarly, once the private queue has been dequeued by the handler the communication is then single-producer single-consumer; the client enqueues calls, the handler dequeues and executes them. Again an efficient queue can be constructed to especially handle this case. These optimizations are important as they are directly on the critical path of all operations between clients and handlers.

5.3.3 Client requests

The handler-loop implementation, above, resides in the run-time library. The client-side is where the compilation and run-time system meet. In particular, the compiler must emit code for the client to package and enqueue requests for the handler, and handle waiting for the results of `separate` queries.

When a client reserves a single handler with `separate h do <body> end`, this corresponds to the code shown in Figure 5.12. The client gets a private queue `h_p` for the desired handler h, represented in the SEPARATE rule by the private queue appearing on the handler's queue-of-queues. This private queue can either be freshly created or taken from a cache of queues, to improve execution speed. It then enqueues this new private queue on the queue-of-queues for the handler, which means the private queue is now ready to log calls on in the body. Finally, corresponding to reaching the end of the `separate` block, the special call denoting the end of the requests on the private queue is enqueued for the handler.

Inside the body of the `separate` block, there will typically be many calls

```
private_queue* h_p = client.queue_for (h);

// SEPARATE rule, adding an empty queue
//   to the queue of queues
h.qoq.enqueue (h_p);

<compiled body>

// SEPARATE rule, compiler adds the
//   code to enqueue the END marker
h_p.enqueue (END);
```

Figure 5.12: A compiled `separate` block

to the handler. The asynchronous calls are packaged by the client using the libffi library [57]. libffi is a high-level interface to various calling conventions; SCOOP/Qs uses it as a way to manage the details of the type and number of arguments, as well as result storage handling. This packaged call is then put into the proper private queue for the desired handler. This can be seen in Figure 5.13, the enqueue operation relating directly to the CALL rule. Packaging the call

```
arg_types[0] = &ffi_type_pointer;
arg_values[0] = &arg;
ffi_prep_cif(ffi_call, FFI_DEFAULT_ABI, 1,
             &ffi_type_void, arg_types);

// CALL rule, showing the setup via libffi.
h_p.enqueue(call_new(ffi_call, 1, arg_values));
```

Figure 5.13: Enqueueing an asynchronous call

entails setting up the call interface (cif) with the appropriate argument and return types with `ffi_prep_cif`, and then storing the actual arguments for later application by the handler. Note that the allocation of arguments and argument types for the call cannot be done on the client's stack because the call may be processed by the handler after the client's stack frame has been popped.

For efficiency reasons, a different strategy is used for synchronous calls (queries). This is because packaging a call involves allocating memory, packing structures properly, and then the handler must decode it when received. In short: this takes longer than a regular function call. In the asynchronous case these steps are required because the execution of the call must be done in parallel with the client's operations. However, for synchronous calls this is no longer the case: the client will be waiting for a reply from the supplier when the supplier finishes executing the query. To address this performance concern, for shared-memory systems, the QUERY rule is changed to the following:

$$\frac{}{\begin{array}{ll}(h, q_h, \mathsf{query}(x, f)) & \| \; (x, q_x, t) \Rightarrow \\ (h, q_h, \mathsf{wait}\ x; f) & \| \; (x, q_x \left[h \mapsto q_x[h] + [\mathsf{release}\ h]\right], t)\end{array}}$$

Note that the execution of the call *f* is shifted to the client, after the synchronization with the handler has occurred. This does not change the execution behaviour because, as in the original rule, all calls on the handler are processed before the query and the client does not proceed to log more calls until the query has finished executing. As can be seen from Figure 5.14, the old rule first

```
<packing same as async>
ffi_call(&ffi_call, f,
         &result, 0);
// QUERY rule
h_p.enqueue(ffi_call);
// SYNC rule
h_p.sync();
```

(a) Generated code for initial SYNC rule.

```
// New QUERY rule
h_p.enqueue(SYNC);
// SYNC rule
h_p.sync();
// New QUERY rule
result = f();
```

(b) Generated code for modified SYNC rule.

Figure 5.14: Executing a query *f*

generates the call, sends it to the handler, and then synchronizes (Figure 5.14a), these actions come from the combination of the QUERY and SYNC rule. The new rule just performs the call after synchronization occurs (Figure 5.14b). This approach offers three main benefits:

- there is no memory allocation required,
- no encoding/decoding of the call is required, and
- which call is being made is known statically.

The last item is important to the underlying optimizer. The LLVM framework now statically knows what call is being made, enabling more optimizations. This isn't the case when asynchronous calls are packaged as a function pointer and arguments: that information is lost for the compiler.

A final scheduling optimization is related to the use of the sync operation: when a sync operation is sent, the client's thread context is sent with the operation to the handler. When the handler encounters the sync, it uses this context to switch immediately to the client. This prevents the lightweight threads from needlessly going to the scheduler. Such an optimization is safe because after sending the sync request to the handler the client must wait. Therefore the handler can have no requests pending after the sync request, because the client is the only entity that can put them there, and it is waiting on the handler.

5.3.4 Wait conditions

The final piece of compiling SCOOP/Qs programs is to process the wait conditions. An example of this situation is given in Figure 5.15. The `require` clause is compiled the same as any query call that a client would want to make on a handler: the handler is reserved, the sync is issued, the query is executed. The difference comes after the evaluation of the query: if the condition is true, then control passes to the body of the `separate` block. If the condition is false, then the protocol in Figure 5.16 is used. This means that the client will wait until

```
separate x
  require
    x > 0
  do
    ...
  end
```

Figure 5.15: A wait condition

```
// Unlock so others can use the handler
unlock(h_p);

// Wait for the wake-up
yield_until_notify(h_p);
```

Figure 5.16: Waiting for a change

awoken by a signal from the handler. However, when awoken the client will check the previous client and only continue if it is different.

The handler loop from Figure 5.11 has to be modified to record the previously finished client and notify any waiters. The modifications, given in Figure 5.17, follow the private queue processing loop.

```
while (private_queue.dequeue (&call))
  {
     execute_call (call);
  }

last_client = private_queue.client;

yielders.notify();
```

Figure 5.17: Modified main handler-loop for yielding clients

5.3.5 Multi-reservation separate blocks

The code that must be generated for the multi-reservation separate block differs slightly from the single-reservation case which is optimized due to it being a simpler operation. One can see in Figure 5.18 that some of the complexity is pushed into the client run-time library. The run-time maintains structures that allow the multiple handlers to be stored. The interface between the compiled code and run-time consists of marking the start of a new set of reservations with `new_reservations`, adding a handler with `add_handler`, and finally reserving all the handlers atomically with `reserve_handlers`. After this point the client can fetch the private queues that were just reserved freely, and they do not need to be inserted into the handler's queue-of-queues because the reservation mechanism has already done that. Signalling the end of the private queue is done as before.

```
client.new_reservations ();
client.add_handler (h1);
client.add_handler (h2);
client.reserve_handlers();

private_queue* h1_p = client.queue_for (h1);
private_queue* h2_p = client.queue_for (h2);

<compiled body>

h1_p.enqueue (END);
h2_p.enqueue (END);
```

Figure 5.18: A compiled 2-reservation `separate` block

5.3.6 Removing redundant synchronization

The SCOOP model essentially prevents data races by mandating that one must access (read and write) separate areas of memory through its respective handler. Due to this, a common SCOOP idiom is that memory is often copied back and forth between handlers when a local copy is desired for speed reasons. One example of this is sending data to a worker for further asynchronous processing. When copying data in SCOOP there are essentially two options: push or pull. Either the data is copied via routines that asynchronously push data to a `separate` target, or the data is synchronously pulled by the client that wants it using queries. Even though the first option appears to enable more concurrency because it is asynchronous, it often isn't the case. Consider sending an array one integer at a time: this involves reading the integer from the client, packaging the call that will set the integer on the handler, sending the call, then applying the call. The speed advantage of utilizing more than one core is dwarfed by the huge cost of issuing the call. Also, the second option (synchronous pull) tends to be more natural, as the client knows how and where to reconstruct the data.

Therefore it is natural to make queries as efficient as possible. This was partially addressed using the approach in the previous section, using sync operations and executing the query on the client. There is a further enhancement that can be made to this approach, which is eliding unnecessary sync calls. A sync call is not necessary if the previous call to the desired handler was also a sync call; basically if the handler is already "synced" it doesn't need to be re"synced".

This elision happens in two ways: either by dynamically recording the synced status in the run-time and ignoring sync operations on handlers that have already been synced, or statically by performing a static analysis.

Dynamic avoidance

The dynamic method keeps the synced status in the private queue structure. When a sync call is made on a private queue, nothing happens if the queue is already synchronized; the call merely returns and the synced status is unaffected. If the queue is not currently synced, the sync message is sent to the handler as usual and when it returns the synced flag is set in the handler reflecting that the handler is processing this private queue, but the queue is empty.

Static removal

The static analysis starts by traversing the control flow graph (CFG). It annotates every basic block, basic blocks being sequences of basic instructions, with a set of handlers that are synchronized by the end of the block. This set of handlers is called the *sync-set*. The traversal of a function's basic blocks can be seen in Figure 5.19. Each block acts as a sync-set transformer, adding and removing

```
while changed $\neq \emptyset$
    b $\in$ changed, changed := changed $-\{b\}$
    common := $\bigcap$ b.predecessors.sync_set

    if b.sync_set $\neq$ UpdateSync (b, common)
        b.sync_set := UpdateSync (b, common)
        changed := changed $\cup$ b.successors
```

Figure 5.19: Sync-set calculation for a function

handlers from the set. As an initial input, the intersection of the sync-sets of all the block's predecessors is used. The traversal continues until every basic blocks' sync-set has stopped changing.

Of course this only says how the blocks are traversed, not how a given block's sync set is calculated given the instruction in that block. This is described in the `update_sync` function, shown in Figure 5.20. Each type of instruction is

```
UpdateSync (b, synced):
    for inst $\in$ b
        h := HandlerOf(inst)
        synced := synced $\cup$ {h}    if inst is a sync operation
                  synced $-$ {h}    if inst is asynchronous
                  $\emptyset$               if inst has side effects
                  synced           otherwise
    return synced
```

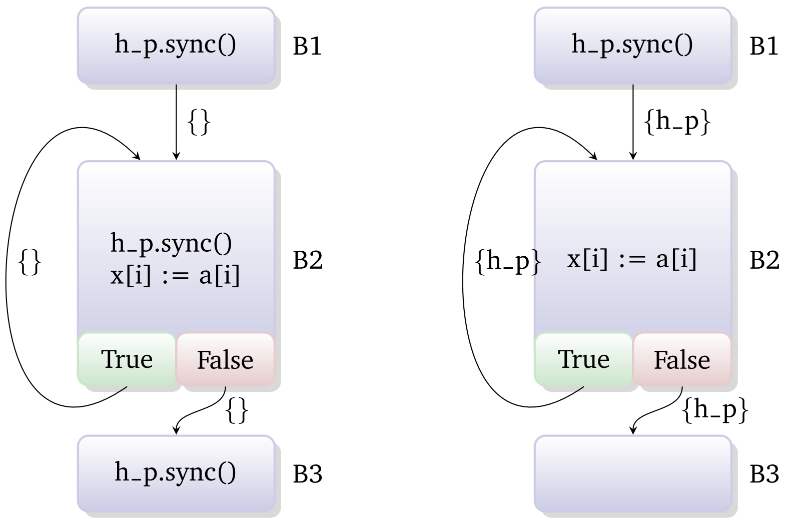
Figure 5.20: Sync-set calculation for a block

handled differently: synchronization calls add the target handler to the sync-set, asynchronous calls remove those handlers (and anything they may be aliased to), and arbitrary calls clear the sync-set entirely. Obviously this final case is quite severe, as it has to be, because a call could subsequently issue asynchronous calls on all the handlers currently in the sync-set. This can be mitigated by not clearing the sync-set for functions which are marked with the `readonly` and `readnone` flags. LLVM will automatically add these flags when it can determine that they hold.

The static analysis operates on LLVM bit-code, and is implemented as a standard LLVM pass (outside of the base compiler). Keeping the pass outside of the base SCOOP/Qscompiler has the advantage that it separates the generation of code from the analysis and transformation of the generated control flow graph.

Example

The effect of the sync coalescing pass can be seen in Figure 5.21. This program



(a) A simple loop before the sync-coalescing pass.

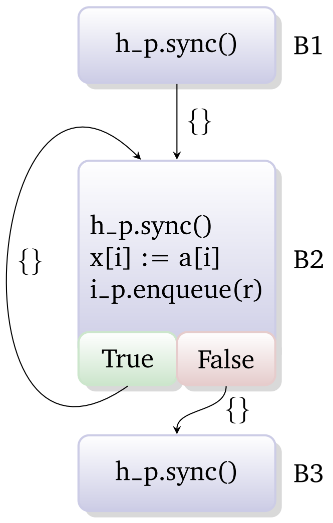
(b) After sync-coalescing sync-sets label edges.

Figure 5.21: Sync-coalescing pass

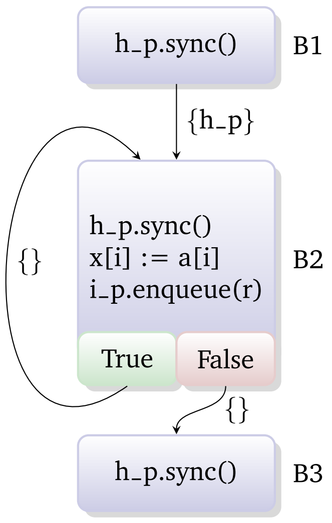
has three blocks, with sync operations in each one. Before the sync-coalescing pass, in Figure 5.21a, the client is reading values out of a handler's array, for which a naïve code generator will produce a sync before every array read. Figure 5.21b shows the results of the sync-coalescing pass in such a situation. The sync-sets are shown explicitly on the edges out of each block. In this case there are no calls that may invalidate a sync-set, so the handler `h_p` appears on all edges. The result of this is that the sync calls in blocks B2 and B3 can be removed. Removing sync calls in the body of a loop can greatly increase performance. Note that even though the sync call in the body of B2 was removed, `h_p` still appears on B2's outgoing edges because B2 doesn't invalidate the synchronization on `h_p` by issuing an asynchronous call.

It is not always possible, however, to remove the sync operations, even when the handler is apparently unaffected. Consider Figure 5.22, where there is an additional call to `i_p.enqueue(r)`, in Figure 5.22a. Enqueueing a call is an asynchronous activity, but it occurs on a different handler variable. This is not enough, though, to conclude the handler `h_p` is unaffected, as these are only variables and could be aliased to one another. Meaning they effectively be the same handler. This means that at the end of the B2 block the outgoing edges are labelled, visible in in Figure 5.22b, with neither `h_p` or `i_p`. If more aliasing information is given to the compiler then it is possible that this ambiguity can be resolved and `h_p` can be added to the sync-set for the block.

The static analysis is important as it goes further towards getting SCOOP out of the way of the optimization passes. In the end, the final implementation



(a) A simple loop with an extra asynchronous call on i.



(b) Handlers `h_p` and `i_p` may be aliased: no coalescing.

Figure 5.22: Ineffective sync-coalescing pass

uses both the static and dynamic approaches. The static analysis is used when it can be, but it is necessarily conservative. For the cases where the static analysis keeps an unnecessary sync operation around, the dynamic check will eliminate the round-trip to the handler.

5.4 Qs run-time design

The implementation of the SCOOP/Qs run-time is organized into layers. Each layer builds on the layer beneath it. This organization was essential to keep the construction of the run-time an approachable activity. The layers also have their own tests that help localize any faults that may appear in the implementation.

The layers can be seen in Figure 5.23.

5.4.1 Task layer

The task layer of the Qs run-time provides basic functionality for task creation and switching. It does not offer any concurrent execution, but allows concurrent execution to be constructed by higher layers.

Contexts

A context in the task layer is the representation of a task's execution information. It consists of any registers that have to be saved, as well as the stack frame pointer. There are operations to

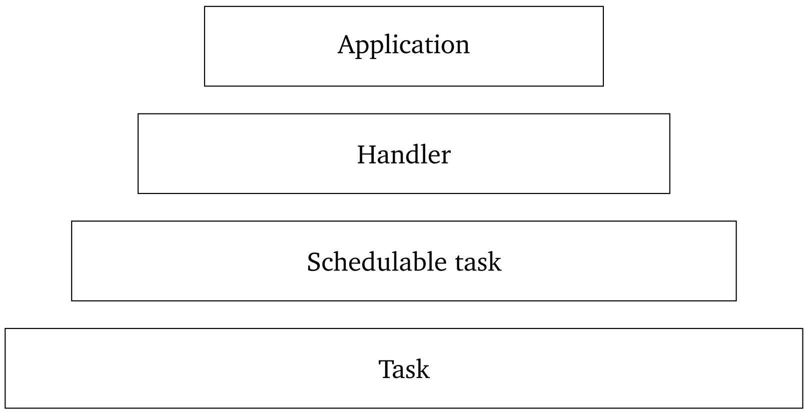


Figure 5.23: Layers of Qs runtime implementation

save store the current register contents and stack so that the context may be resumed later.

resume resume the given context. This operation does not return, as it immediately transfers to the context.

set_next set the context to switch to if the current context exits. This is useful to define a cleanup routine.

Contexts are currently implemented internally using `jmpbuf` and the associated `siglongjmp` and `sigsetjmp` routines, as it is the most accessible implementation of such a context-saving module. More efficient implementations would still be possible under the context interface, but so far the context switching mechanism has not been identified as a bottleneck to performance.

Tasks

Tasks are a combination of context and state. The context allows computation to be suspended and resumed, where the state indicates if the task is waiting, running, or runnable. There are also states to represent transition between states, such as transition-to-waiting. The transitional states are used to indicate when the task is transitioning between states, which is important when tasks are running concurrently and the responsibilities for task movements are not centralized. In this case, it may be that as a task is transitioning to the waiting state it is dequeued and asked to wake up again. The task must not be in both a wait queue and in a run queue. The Linux kernel uses a similar mechanism, although it does not use transitional states, it uses an `on_cpu` flag to denote if the task is still being processed, which can be used to derive if a task is still transitioning its state to waiting. The Qs runtime uses explicit transitional states to provide a more explicit representation of the system state.

The task states are similar to those in the Linux threading model, but more explicit with the inclusion of a runnable and transitional states. The transitions between states can be seen in Figure 5.24.

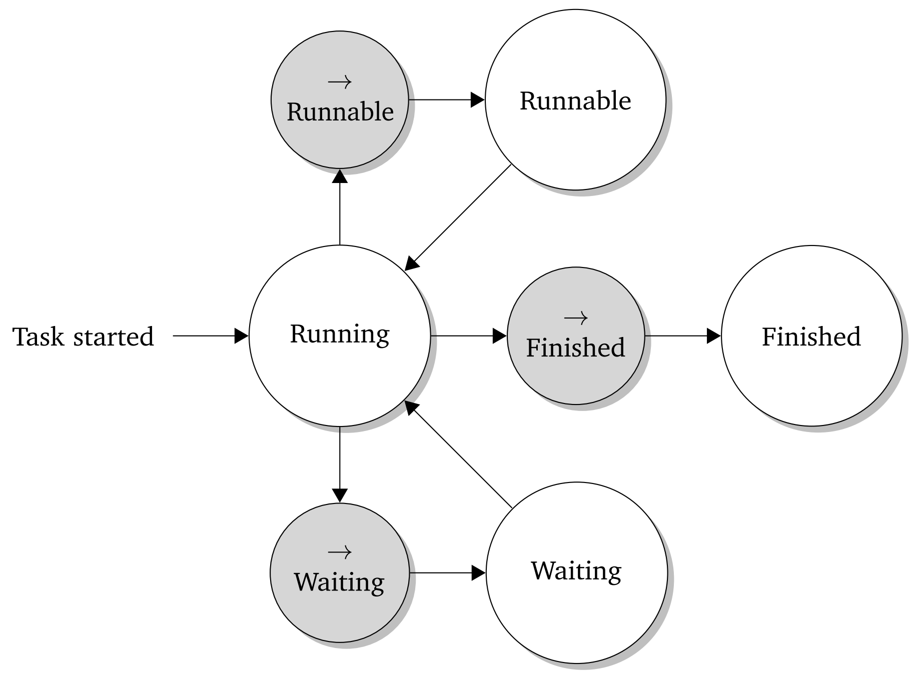


Figure 5.24: Task state transition diagram

5.4.2 Schedulable task layer

The schedulable task layer is responsible for the scheduling of tasks. To do this it introduces executors, global operations, schedulable tasks, and basic synchronization primitives.

Executor

The executor is a local scheduler that is responsible for holding a list of tasks that are to be scheduled. These tasks are held in a lock-free deque, otherwise known as a work stealing queue. The executor will first try to schedule tasks in its own work queue, if its own queue is empty, it asks the global data for other executors and will try to steal from their work queues, if possible. If this also fails, it waits on the global work queue for more tasks. Note that only the tasks that belong to executor can push work into its work stealing queue, although any other task or executor can steal from it.

The executor is itself a task, and thus works on the same principles as other tasks. The distinguishing feature of an executor as a task is that it never exists in any task queues: no sleep-queues, run-queues, wait-queues. It is the target of a task switch when a task yields, which they do periodically.

Schedulable tasks

Schedulable tasks are an extension of the tasks in the “vanilla” task layer. They are able to interact with the executors and the global synchronization information. Schedulable tasks are additionally responsible for performing the state transitions on other schedulable tasks that may be incoming.

This represents a fundamental design of the SCOOP/Qs runtime: that schedulable tasks are able to attempt to schedule *other* tasks themselves, without encountering any operating system locks, or context switching to the executor. If this fails (i.e., no tasks are available), then the responsibility of scheduling the task is given to the executor which will wait if necessary.

Global operations

There are operations that operate on global data. These are required to coordinate executors with one another, and also for the basic start-up and shutdown of the SCOOP/Qs run-time.

create executors create a given number of executors at program start-up.

join executors wait for all executors to finish. This is used in the setup routine for a program so that the program only ends when all executors have finished processing tasks.

signal work signal that tasks are available in the global work queue or in the work queue of an executor.

wait for work used by executors to wait for new runnable tasks to be available in either the global queue or a work stealing queue of another executor.

enqueue runnable enqueue a new runnable task in the global run queue.

try dequeue runnable a non-blocking operation that tries to dequeue tasks from the global run queue.

Synchronization primitives

Since this layer introduces preemptive concurrency, it is beneficial to take the unit of concurrency, the schedulable task, and reconstruct basic synchronization primitives in terms of the multitasking operations provided by this layer. Re-implementing these common synchronization primitives has two benefits:

1. it simplifies the implementation of some basic operations because they do not have to manually schedule themselves,

2. it avoids the use of "heavy" operating system primitives which, if they block, would occupy an entire executor and it would be unavailable to execute more tasks.

Even though the highest layer (SCOOP code) will not use these primitives, they are essential to the implementation for both abstraction and performance reasons.

There are two primitives available, the mutex and the condition variable. Other primitives could be added as required, although as of yet there has been no need identified within the runtime implementation for anything beside these primitives.

5.4.3 Handler layer

Private queues

The private queue structure enables communication between the client and handler. It has few, but important, operations:

new Constructs a new private queue. Its only argument is the handler with which it is supposed to communicate. Private queues are not created directly, but only when a client asks for a new private queue for a given handler. Clients contain a cache of handler so that a new private queue does not have to be created every time. This requires an extra lookup in a hash-table, but this cost is generally less than constructing a new private queue.

The client of the private queue is not stored, it is only known implicitly as the private queue will be in the cache of its respective client.

The code generation and run-time must be cautious to make sure only a single client uses the private queue, which is why whenever a private queue is needed the client uses its private queue cache.

lock enqueue this private queue in the handler's queue-of-queues.

unlock enqueue a message in the private queue that tells the handler that this private queue is done for the moment.

sync send a sync message to the handler and wait for a response.

dequeue used by the handler to ask for a new message from this queue.

is_synced a query which will say if the client is currently synced with the supplier. This is stored as a Boolean flag, remembering if the last operation was a sync.

new_reservations Mark the beginning of a new set of reservations, yet to be added.

add_handler Adds a handler to the set of reservations.

reserve_handlers Actually reserves the set of reservations. This uses spinlocks on the handlers to assure mutual exclusion.

Also, since private queues only ever have a single thread enqueueing message and a single thread dequeueing, then it falls into the single-producer single-consumer pattern (SPSC). This allows the usage of a specialized data structure: the SPSC lock-free queue. SPSC lock-free queues are more specialized than the general case of multiple-producer multiple-consumer (MPMC), which means they are only correct in situations where SPSC applies. For this loss in flexibility, they gain in efficiency. In particular, on x86 architectures they can be implemented without any assembly-level lock instructions, which is the implementation that is used in SCOOP/Qs.

The multiple handler reservation is currently implemented using spinlocks. However, since enqueueing in the queue-of-queues has a very efficient implementation on shared-memory systems (a single atomic exchange operation, plus one or two regular memory read/write operations) this could be a candidate to be rewritten using software or hardware transactional memory.

Queue-of-queues

Each handler owns a unique queue-of-queues, which no other handler will dequeue calls out of. The queue-of-queues follow a multiple-producer single-consumer discipline, which again means that a specialized lock-free queue implementation is used for efficiency. Queue-of-queue structure is even simpler than the private queue because it does not require the ability to remember if it is synchronized with any clients, nor does it have to remember its handler or any of its clients.

Handler

The handler is the main actor in the SCOOP model. Handlers enqueue private queues, into which they will later log calls, into other handler's queue-of-queues. Likewise when idle they look in their own queue-of-queues for new private queues which have to be processed. Therefore, every handler must contain a queue-of-queues.

However, to support the full SCOOP capabilities, other support structures are needed. The existence of wait-conditions requires handlers to have a notification mechanism when the state of the handler's owned objects may have changed. To achieve this, the handler also has a mutex, condition variable, and a reference to the last client that the handler executed requests for. All together, these allow the implementation of a notification system that will allow a client to wait for change notification from a handler, but not be notified of its own queries. When finished processing a private queue, the handler will set the last client variable and signal the condition variable. Those waiting on the handler will awake and check that the last client variable is not them, and if so try to execute their separate query again. The client continues to wait if the last client variable still refers to them.

The operations in the lifetime of a handler are visualized as a state transition diagram, appearing in Figure 5.25. A handler starts out its life receiving private queues from its queue-of-queues. When a private queue is available, it is taken and calls received from the private queue. Each call is then applied and the handler continues this process of taking calls and applying them until the end of the private queue is encountered. The end of the queue is not symbolized by the private queue being empty, but rather is a special message that will be sent through the private queue. When the private queue is finished, the handler will notify other clients that may be waiting on it that its state may have changed and they should retry their wait-conditions. After this, it returns to processing from its queue-of-queues until it is told to shut down, which is an externally initiated event. This shutdown can occur manually as in the SCOOP/Qs prototype, or by the garbage collector when it determines the handler is no longer needed.

5.4.4 EVE run-time comparison

The techniques presented for SCOOP/Qs have also been integrated into the research branch of the EiffelStudio IDE and compiler. This implementation expands on the semantics in Section 5.2 to include reserving and yielding on multiple handlers, rather than the single-handler version that was used in the simplified semantics. There is an implementation of the SCOOP run-time already in production; this section compares implementation differences between the

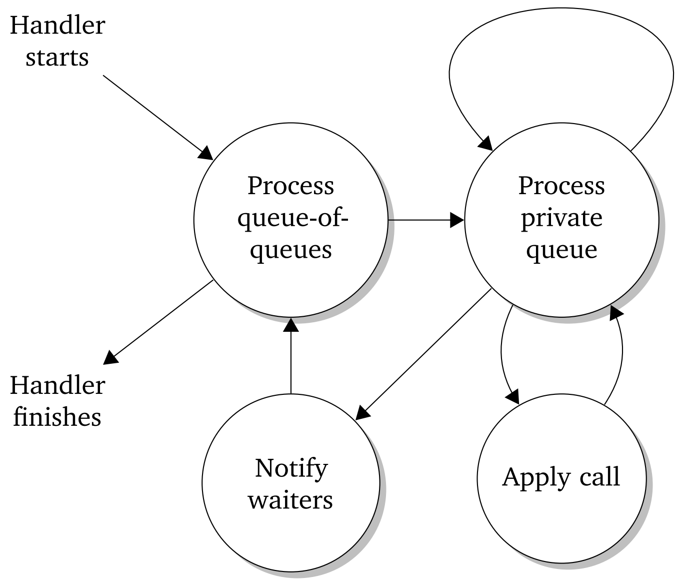


Figure 5.25: Handler state transition diagram

production SCOOP run-time and the run-time crafted for EiffelStudio using SCOOP/Qs techniques.

Core implementation differences

The SCOOP run-time used in the production EiffelStudio compiler is written mostly in Eiffel itself. This has advantages, such as a garbage collector and memory safety. However, depending on the program, during execution the facilities provided by the run-time may be called very many times. Because of this, the EVE/Qs run-time is written in C++ instead of Eiffel. This decision contributes to the efficiency of the EVE/Qs run-time.

Additionally, the SCOOP/Qs run-time uses lightweight threads. The EiffelStudio system, instead, uses system threads (i.e., POSIX threads). The system thread SCOOP implementation in EiffelStudio has two main disadvantages: the cost of context switching is comparatively high due to the system threads, and the use of thread local storage to store handler context data means that it is harder to differentiate between handler and thread.

Compatibility

The EVE/Qs run-time is nearly a drop in replacement for the existing EiffelStudio run-time, with two basic exceptions. The first exception is the handling of wait-conditions, which in the EiffelStudio run-time are periodically re-evaluated to see if the result had changed, and in EVE/Qs this is not done because the choice of timeout offers only another parameter to tweak. In addition, the situations where this lack of timeout becomes noticeable is when the precondition relies on the passing of time, or the execution of the precondition itself, to become true.

Neither of these situations are supported by the EVE/Qs run-time. However, one may argue that these can either be designed around (for example using a timer to trigger the condition explicitly rather than implicitly), or are bad designs to begin with.

The interface of EVE/Qs is mostly compatible with that of EiffelStudio. As EiffelStudio compiles down into C, this means that the same C code is generated whether the stock run-time or EVE/Qs run-time is used. The difference is hidden in the common C header files which can switch between the two implementations given different C compiler flags (defining different constants for the pre-processor). This means that supporting both code paths is easily accomplished; the choice of run-time can be made with a compilation switch.

Allocation difficulties

One of the largest hurdles to overcome in making the EVE/Qs implementation perform better than the existing solution is the implementation of the EiffelStudio memory management system. The memory allocator and collection system is heavily guarded by mutexes. Unfortunately, these mutexes are global and are required for even the most basic SCOOP operations, such as when clients package the function pointer and arguments to log calls for a handler. One of the important improvements of EVE/Qs was to fall back on the system malloc implementation to allocate these structures.

However, since these structures may pack away references to objects allocated by the normal run-time, they must be traversed during garbage collection. If these structures were not traversed, then the collection may miss references to live objects. Due to this, the EVE/Qs includes marking routine that is conditionally called during the standard garbage collection cycle which will make sure that any references that are "hidden" in the EVE/Qs run-time are made known to the stock Eiffel run-time.

These allocations also occur when setting up return points for exception handling. Exception handling happens in the EVE/Qs run-time when a handler executes a call on behalf of a client. If that call throws an exception, the handler must record it for later reporting back to the client. Since the stock Eiffel run-time's exception catching mechanisms require memory allocations (and these allocations take a global lock), this is another performance bottleneck. Unfortunately, a solution to avoid this allocation has not yet been found.

The introduction of a more modern garbage collection system for EiffelStudio would alleviate these concerns and obviate the need for the, currently necessary, mitigation mechanisms.

Impersonation

One of the fundamental optimizations, client-side query execution, requires additional mechanisms in the Eiffel run-time to work properly. The issue is that when a client runs a query instead of the handler, that query still has to believe it is being executed by the handler. In the SCOOP/Qs run-time this is trivial because the handler context is passed around explicitly and the client-side query execution merely has to pass the appropriate handler's context to the query. However, in EiffelStudio the SCOOP context is kept in thread local storage. This means that the handler context is bound to the thread, and it client must update the thread local storage to reflect with handler's context before the client-side query is called, and restore its own context after the call finishes. This reduces the simplicity of the optimization when compared with the approach that is taken in the SCOOP/Qs run-time.

5.5 Related work

Finding run-time and compiler optimizations is a vital research goal when developing programming approaches for concurrency and parallelism. While approaches in this area are based on a broad variety of concepts, and in this respect each require different solutions, this work profited from insights and discussions of a number of related works.

For the SCOOP model there are different approaches to improve the performance of execution. One approach called *passive processors* [72] turns `separate` objects into simple lock-protected objects, their handler no longer processes the calls and leaves it to the client. This mechanism is like a manually enabled instance of client-side query execution, but can also enable client-side routine execution. Additionally an approach which divides arrays among multiple handlers [92] allows for competitive workloads on arrays. This is done by disabling the SCOOP data race freedom guarantees for some library routines which will divide up parts of the array so they do not have to be copied explicitly from one handler to another. This is similar, although done at a lower level, to [49]. SCOOP/Qs, to contrast with array slicing, does not revoke any data race freedom guarantees.

Cilk [10] is an approach to multi-threaded parallel programming based on a run-time system that provides load balancing using dynamic scheduling through work stealing. Work stealing [11] assumes the scheduling forms a directed acyclic graph. In contrast, SCOOP/Qs tolerates some cyclic schedules through the use of queues. Since SCOOP/Qs use queues, handler A can log work on handler B while handler B logs work on A, as long as they do not issue queries on one another (forcing a join edge). Also, SCOOP/Qs is not strict: edges go into handlers from the outside, other than at spawn; this is actually the normal case when logging calls. Although Cilk has been extended into Cilk++ [34], this does not indicate a significant uptake of object-oriented concepts to ensure correctness properties such as race freedom.

X10 [18] is an object-oriented language for high performance computing based on the partitioned global address space model, which aims to combine distributed memory programming techniques with the data referencing advantages in shared-memory systems. Although there is a mechanism to ensure local atomicity through the keyword `atomic`, it is opt-in, and as such admits racy programs by default. The `async` blocks allow computations to run on different address spaces, but there is no way for the caller to ensure consistency between `async` blocks directed to the same address space. The help-first stealing discipline [39] in X10 offers that the spawned task is left to be stolen, while the worker first executes the continuation; this is in contrast to Cilk's work-first strategy where the spawned task is executed first. The help-first strategy has benefits as it avoids the necessity of the thieves synchronizing. This only applies because the thefts in a finish block in X10 are serialized in work-first, whereas they are not for help-first. This technique would not be directly applicable to SCOOP/Qs because

SCOOP/Qs waits only on the result of a single handler.

Aida [61] is an execution model that, like SCOOP, associates threads of control with portions of the heap. The technique is implemented on top of Habanero-Java [16], an extension of the X10 implementation for Java. When there is contention for a particular heap location, the "loser" rolls back its heap modifications, suspends, and appends itself (delegates) to the run queue of the winner, effectively turning two concurrent tasks into a single one. This is fundamentally different from the SCOOP model, which also has isolated heaps, but allows interaction between threads of control, and even provides reasoning guarantees on this interaction. Therefore the underlying mechanisms are fundamentally different, where Aida requires efficient heap ownership and conflict resolution via a parallel union-find algorithm, SCOOP/Qs requires efficient communication which is attained via novel and nested uses of specialized queue structures. Otello [113] extends the isolation found in Aida to include support for nested tasks.

Another object-oriented approach which, like SCOOP, associates threads of execution with areas of the heap is JCoBox [91]. It also makes the distinction (similar to `separate`) between references that are local and those that are remote, although this can only be applied per-class, not per-object as in SCOOP. Each CoBox contains a queue for incoming asynchronous calls, though the reasoning guarantees are weaker for JCoBox, so this structure can be simple. The synchronous calls in JCoBox are also executed locally, but no dynamic or static method to reduce communication, ensuring data race freedom, is performed.

Kilim [97] is a framework that supports the implementation of Actor-based approaches in Java. It improves message-passing performance by treating messages differently from other Java objects, in that they are free of internal aliases and owned by at most one Actor at a time. The messages arrive via explicitly declared mailboxes in the objects, which also do not provide the reasoning guarantees between messages that the SCOOP model provides. The Kilim mailboxes have, therefore, a more simplistic behaviour compared to the queue-of-queues approach in SCOOP/Qs. Kilim also sets new standards in creating lightweight threads, which are not tied to kernel resources, thereby providing scalability and low context switching costs. SCOOP implementations have previously been based on operating system threads, and using lightweight threads in SCOOP/Qs offers similar improvements in scalability as observed by Kilim.

Kilim is extended with ownership-based memory isolation [37] for messages to reduce the amount of unnecessary copying. Although not strictly a message-based model, SCOOP/Qs may be able to apply this technique to so-called expanded classes, which are more like standard C structures, and are presently copied when used as arguments to separate calls.

The above is summarized by stating whether they offer *guards* (protection against races) and *delegation* (ability for one entity to give work to another).

- *No guarding, no delegation* – Cilk/Cilk++.
- *Partial guarding, delegation* – X10 allows delegation as the only way for one place to modify another. However, a place can asynchronously modify itself through the same mechanism, thus there may be races within a place.
- *Guarding, protective delegation* – Aida and Otello extend X10 with the ability to resolve races by rolling back changes and reducing the amount of

concurrent execution.

- *Guarding, delegation* – JCoBox and Kilim both have different approaches to the actor/active object model. This implies strict guarding and delegation of actions.
- *Guarding, enhanced delegation* – SCOOP follows the actor approach, but then also offers enhanced delegation by allowing clients to maintain pre/-postcondition reasoning with the handlers that they are delegating to.

5.6 Conclusion

This work focuses on SCOOP/Qs, an efficient execution model and implementation for the SCOOP concurrency model. As many other programming models that ensure strong safety guarantees, SCOOP introduces restrictions on program executions, which can become performance bottlenecks when implemented naively, standing in the way of practicality and more widespread adoption. The key to the present approach is a reexamination of the SCOOP guarantees, allowing one to explore a larger design space for run-time and compiler optimizations. In particular, it enabled the removal of much of the need for synchronization between threads, thereby providing more opportunities for concurrency.

The underlying techniques used in SCOOP/Qs are an efficient way to offer temporary control of one active object, or actor, over another. As such the technique could also be used in approaches like JCoBox [91] or Kilim [97] to provide more structured interactions between entities.

The overall approach given here, without the detailed run-time design discussion, is also available from [107].

Chapter 6

Evaluation and comparison

To provide confidence that the design and implementation of SCOOP/Qs really provide increased performance, the performance must be measured. This evaluation aims to thoroughly explore the performance characteristics of the SCOOP/Qs techniques. To do this, SCOOP/Qs is evaluated on a variety of benchmark programs, and is compared in different ways. The comparisons are:

optimization the basic implementation is controlled and the optimizations that are used are compared

implementation different implementations of the SCOOP language are compared

language different languages are compared on the benchmarks

These comparisons work from the most specific (optimization) to the most general (language). This style allows for a great deal of insight into the specific and general characteristics of the SCOOP/Qs approach.

Section 6.1 gives an overview of the general categories of programs that are benchmarked, the specific benchmark programs, and the testing procedure. Section 6.2 shows the effect of applying different SCOOP/Qs optimization techniques on the performance. Section 6.3 compares against the existing EiffelStudio implementation against the implementation of the ideas in EVE (EiffelStudio's research branch), and also a fresh implementation that can do more aggressive optimization. Section 6.4 shows the performance of SCOOP/Qs along side a broad variety of other paradigms and languages for parallel and concurrent programming – C++/TBB, Go, Haskell, and Erlang – demonstrating competitiveness of SCOOP/Qs in a wider context. Section 6.5 gives a review of other benchmarking techniques, and Section 6.6 concludes.

6.1 Benchmarking

Properly evaluating a core run-time mechanism, such as SCOOP/Qs, requires that it be used in a diverse assortment of situations. With this in mind, we categorize the benchmarks that will be used into two main groups:

- *parallel*: problems where concurrency is not part of the functional specification, but can be used to speed up the execution.
- *concurrent*: problems which are defined by their concurrent behaviour.

The first work type, parallel, is often a data processing task, where multiple threads each process part of a large data set to decrease the total running time. The second type of work, concurrent, is more about the coordination between the threads of control. Here, the concurrency is part of the system's specification, such as a server handling multiple clients simultaneously.

6.1.1 Parallel

The benchmark programs selected for the parallel problems come from the Cowichan problem set [109]. They focus on numerical processing and working over large arrays and matrices. The programs include:

- randmat: randomly generate a matrix of size n_r.
- thresh: pick the top p% of a matrix of size n_r and construct a mask.
- winnow: apply a mask to a matrix of size n_r, sorting the elements that passed the mask based on their value and position, and taking only n_w from that sorted list.
- outer: constructing a matrix and vector based off a list of points.
- product: matrix-vector product.

These benchmarks can be sequentially composed together, the output of one becoming the input to the next, to form a chain. This chain is more complex and sizable than the individual and gives a more diverse picture of a language's parallel performance.

6.1.2 Concurrent

The concurrent problems focus on the interaction of different independent threads with each other. Three benchmarks represent different interaction patterns:

- mutex: n threads all compete for access to a single resource, the threads do not depend on each other.
- prodcons: n producers and n consumers each operate on a shared queue; the queue has no upper limit so producers do not depend on consumers, but consumers must wait until the queue is non-empty to make progress.
- condition: n "odd" and n "even" workers increment a variable from an odd (even) to an even (odd) number. Each group depends on the other to make progress.

All of the above are repeated for m iterations. Finally to this two concurrency benchmarks from the Computer Language Benchmarks Game [24] are added:

- threadring: threads pass a token around a ring in sequence until the token has been passed n_t times.
- chameneos: colour changing "chameneos" mate and change their colours depending on who they mate with. This is done n_c times.

The combination of these parallel and concurrent benchmarks gives us a balanced view of the performance characteristics of the approach.

6.1.3 Setup

All benchmarks were performed 20 times on a Intel Xeon Processor E7-4830 server (4 $\times$ 2.13 GHz, each with 8 cores; 32 physical cores total) with 256 GB of RAM, running Red Hat Enterprise Linux Server release 6.3. Language and compiler versions used were: gcc-4.8.1, go-1.1.2, ghc-7.6.3, erlang-R16B01. For the parallel benchmarks, the problem sizes used are n_r = 10,000, p = 1 and n_w = 10,000; for the concurrent benchmarks n = 32, m = 20,000, n_t = 600,000, and n_c = 5,000,000.

6.2 Optimization comparison

SCOOP/Qs is a collection of different techniques to optimize the execution of SCOOP programs. It is important to understand what exactly the effect, if any, each optimization has on making the execution faster. This knowledge can be used to decide if a particularly complicated optimization is worth the cost of refined implementation and maintenance.

Here the impacts of the optimizations outlined in Section 5.3 and Section 5.4 are examined. The following optimization configurations are examined in depth:

- Applying no optimizations (**None**).
- Dynamically coalescing sync operations by recording and checking the synchronization status in the runtime (**Dynamic**), as in Section 5.3.6.
- Statically determining unnecessary sync operations and removing them in a compiler pass (**Static**), as in Section 5.3.6.
- Usage of the queue-of-queues and private queues as a handler/client communication abstraction (**QoQ**), as seen in the semantic model given in Section 3.2.
- Applying all of the above SCOOP/Qs optimizations (**All**).
- Using all optimizations, and in addition using Thread-Caching Malloc (TCMalloc) [98] (**All/TC**). TCMalloc is built to be especially efficient when multiple threads are all allocating, which happens very often in the SCOOP/Qs runtime for queue slots and closures. TCMalloc is not an optimization contributed by this work, rather the contribution is recognizing that concurrent memory allocation may be a bottleneck. This is the configuration used later when comparing implementations and languages.

These configurations are compared to evaluate if and when the optimization is most effective.

6.2.1 Parallel

The idiomatic way to transfer data in SCOOP/Qs is to have the client pull data from the handler. This happens in the Cowichan problems often, as the underlying data structures are almost exclusively large arrays and the data in the arrays must be distributed to and from workers. This is important background for understanding the optimization comparison below.

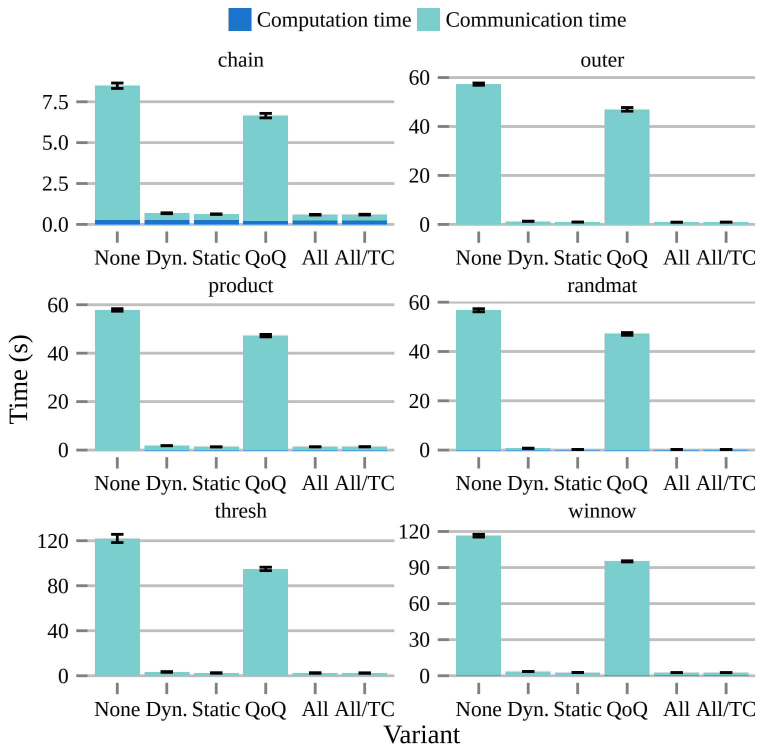


Figure 6.1: Comparison of SCOOP/Qs optimizations on parallel benchmarks with 32 cores

Execution time. Figure 6.1 displays the both the computation and communication time. Computation time is the time spent calculating the result, and the communication time is the difference between this and the total time. The communication time measures the time processors spend waiting for one another and sending data back and forth. From Figure 6.1, it is clear that communication time is a dominating factor in general for the SCOOP programs. It is particularly bad for the **None** and **QoQ** optimizations, and, compared to those, relatively good for the others. This implies that a large portion of the time is spent sending and waiting for data via queries, which the **Dynamic** and **Static** optimizations reduce greatly. This has slightly less of an effect on the chain benchmark because this benchmark communicates less than the others because many of the stages of work are fused together so the data stays on the workers longer.

The difference between having no reduction in the number of sync calls (**None**, **QoQ**) and employing some reduction technique (**Dynamic**, **Static**, **All**, **All/TC**) is that the latter is at least 10 times faster (chain), around 100 times faster (thresh, winnow, outer, product, randmat).

Additionally, since many of the access patterns in these benchmarks are very structured (nested loops for copying matrices), the **Static** removal of sync operations is significantly more effective than the dynamic removal. This is because, seen when examining the generated assembly code, the static removal allows such tight-copy loops to use vectorized instructions (such as SIMD) that

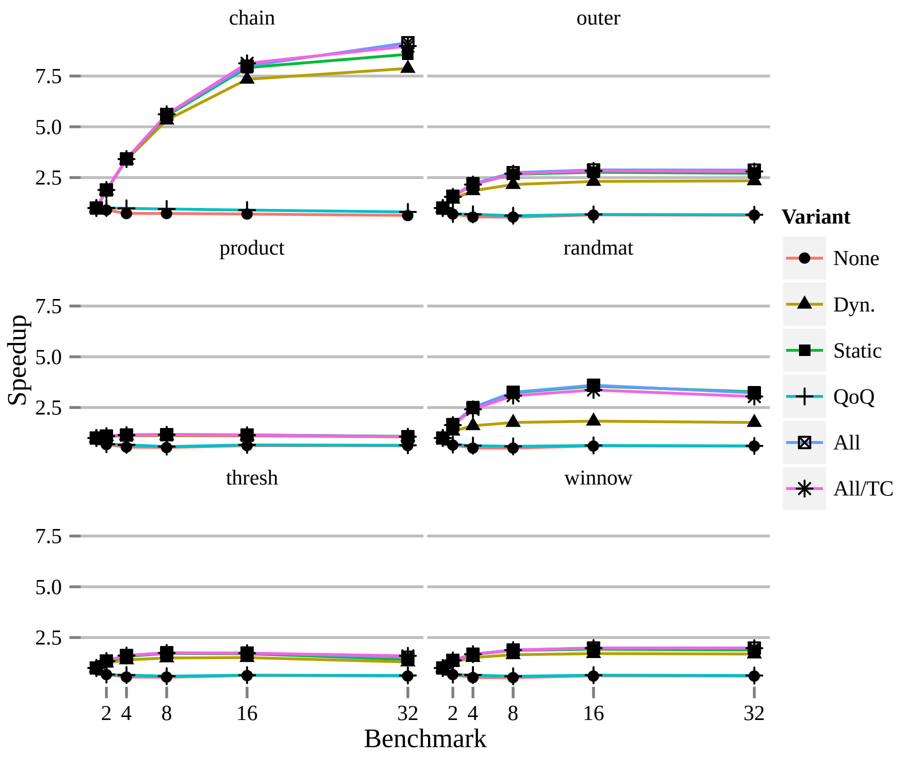


Figure 6.2: Comparison of optimizations on parallel benchmarks' scaling

result in code that looks like a straight memory copy operation such as in `memcpy`.

Although having less of a reduction, the **QoQ** optimization also improves performance slightly (about because it is able to keep the processors running more concurrently. The improvement is typically about 26% over **None**.

The geometric means of all benchmarks for each configuration are **All/TC** (1.32s), **All** (1.33s), **Static** (1.35s), **Dynamic** (1.90s), **QoQ** (38.91s), and **None** (49.14s). One notes by looking **All** and **All/TC** that although effective by itself, **QoQ** does not contribute greatly once **Static** sync reduction is used.

Multicore scaling. The scaling behaviour in Figure 6.2 basically comes in two flavours: those with some form of sync coalescing and those without. When some sync reduction mechanism is used *some* speedup can be seen when increasing the number of cores, although it is clearly not always very large. The best scaling is the chain benchmark which has the least amount of communication and it scales up to 9× at 32 cores. The scaling performance of every Cowichan problem without sync coalescing exhibits a slowdown as the cores increase. This slowdown is typically about 1.5× at 32 cores. This is not entirely surprising as the threads are constantly switching from handler to client trying to get large amounts of data back and forth. Table 6.1 shows the exact times for each number of cores tested.

Task	Optim.	1	2	4	8	16	32
chain	None	5.38	5.95	7.32	7.34	7.66	8.47
chain	Dyn.	5.37	2.89	1.59	1.01	0.73	0.68
chain	Static	5.36	2.86	1.57	0.97	0.68	0.63
chain	QoQ	5.38	5.37	5.47	5.65	5.99	6.67
chain	All	5.38	2.85	1.57	0.96	0.67	0.59
chain	All/TC	5.38	2.85	1.58	0.96	0.66	0.60
outer	None	36.96	52.86	65.20	67.45	55.72	57.55
outer	Dyn.	2.97	2.08	1.61	1.38	1.29	1.27
outer	Static	2.55	1.63	1.18	0.96	0.92	0.94
outer	QoQ	31.26	44.12	45.34	50.23	45.66	46.76
outer	All	2.53	1.60	1.15	0.92	0.88	0.88
outer	All/TC	2.53	1.64	1.18	0.94	0.90	0.91
product	None	35.69	52.21	64.99	67.21	57.20	57.81
product	Dyn.	1.91	1.78	1.72	1.72	1.73	1.81
product	Static	1.44	1.31	1.25	1.23	1.25	1.34
product	QoQ	30.21	43.40	45.68	52.42	45.83	47.10
product	All	1.43	1.31	1.25	1.23	1.25	1.35
product	All/TC	1.45	1.32	1.26	1.25	1.27	1.36
randmat	None	34.08	51.58	67.09	67.55	55.96	56.73
randmat	Dyn.	1.14	0.85	0.71	0.64	0.62	0.65
randmat	Static	0.71	0.42	0.28	0.22	0.20	0.22
randmat	QoQ	28.75	42.66	46.05	48.91	45.36	47.07
randmat	All	0.71	0.43	0.28	0.22	0.20	0.22
randmat	All/TC	0.72	0.44	0.30	0.23	0.21	0.24
thresh	None	74.11	108.51	134.72	136.61	117.49	123.14
thresh	Dyn.	4.37	3.50	3.12	2.92	2.89	3.37
thresh	Static	3.57	2.69	2.27	2.08	2.10	2.51
thresh	QoQ	59.68	88.17	91.91	99.88	92.40	94.54
thresh	All	3.66	2.72	2.27	2.09	2.11	2.40
thresh	All/TC	3.68	2.74	2.28	2.11	2.12	2.30
winnow	None	70.15	105.04	132.83	133.52	114.13	116.33
winnow	Dyn.	5.99	4.62	3.95	3.62	3.51	3.55
winnow	Static	5.13	3.74	3.08	2.75	2.66	2.72
winnow	QoQ	59.84	88.82	91.55	101.12	92.49	95.14
winnow	All	5.10	3.71	3.04	2.70	2.57	2.58
winnow	All/TC	5.12	3.73	3.05	2.72	2.59	2.59

Table 6.1: Times (in seconds) with optimizations applied on parallel benchmarks

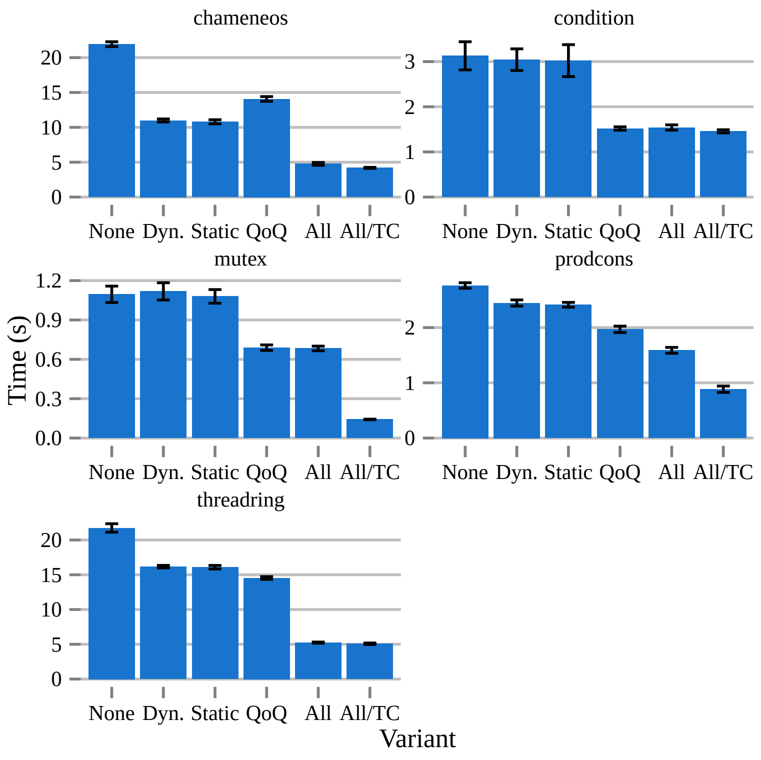


Figure 6.3: Comparison of SCOOP/Qs optimizations on concurrent benchmarks

6.2.2 Concurrent

The effect of the different optimizations on the concurrent problems can be seen in Figure 6.3. One notes that the relative effectiveness of the sync reduction optimizations and the **QoQ** optimization, compared to **None**, have swapped places. While the **QoQ** optimization had an improvement of about 26% on the parallel benchmarks, the effect is slightly greater for the concurrency benchmarks with an improvement of 60%. The increase in performance can be attributed to the better utilization of available processing capabilities resulting from less blocking in the underlying semantics. When using **QoQ** there are also fewer context switches, since the private queues require only one context switch to wait for a query to return. When not using the queue-of-queues a client could wait three times: first for the lock on the handler, then the handler must wait for the client to log its query, then the client must wait again for the handler to return the query.

Instead now the sync reduction optimizations, **Dynamic** and **Static**, now have an average improvement of 25% over **None**. This is a significant improvement; however the concurrency benchmarks do not benefit as greatly from this optimization as the parallel benchmarks do. The sync reduction in particular does not factor greatly into the mutex and condition benchmarks. In the case of mutex syncs are not used at all, and in the case of condition they are only used a little, and the other overheads of scheduling drown out any improvements.

The threadring benchmark is particularly interesting as it shows one case

Task	None	Dyn.	Static	QoQ	All	All/TC
chameneos	21.93	10.99	10.80	14.08	4.76	4.19
condition	3.13	3.04	3.02	1.52	1.54	1.46
mutex	1.10	1.12	1.08	0.69	0.68	0.14
prodcons	2.76	2.45	2.41	1.97	1.59	0.88
threadring	21.74	16.17	16.09	14.54	5.24	5.08

Table 6.2: Times (in seconds) with optimizations applied on concurrent benchmarks

where the improvements seen in **All** and **All/TC** are not the sum of the improvements in the other optimizations. Each of the sync reduction and **QoQ** optimizations reduce the running time by about 5s, but when they are combined the running time drops by 15s. This may be due to the sync coalescing making for shorter reservation times on the handler, and the **QoQ** optimization allowing for more concurrency, meaning in the end there is less overall contention.

Lastly, the use of TCMalloc in the **All/TC** configuration helps the prodcons and mutex significantly. prodcons improves by 2× and mutex sees more than a 4× improvement. These have highly asynchronous behaviours where some workers are constantly logging asynchronous calls which requires rapid small memory allocations on different threads, and TCMalloc helps with exactly this use case.

The geometric mean of all the concurrent benchmarks are **All/TC** (1.31s), **All** (2.11s), **QoQ** (3.36s), **Static** (4.24s), **Dynamic** (4.30s), and **None** (5.38s). The full results are visible in Table 6.2.

6.2.3 Summary

Each optimization has particular situations in which it brings the most benefit:

- **QoQ** is best on coordination tasks but is not as useful for query-heavy workloads.
- **Dynamic** sync-coalescing is absolutely essential tasks with many queries, and shows a good improvement on the concurrent workloads.
- **Static** sync-coalescing is primarily effective on very structured query usages, beating even **Dynamic** in such situations. Similar to **Dynamic** it also improves concurrency benchmarks nicely.
- **All/TC** shows that memory allocation can be a bottleneck due to many small allocations from different threads; TCMalloc helps address this bottleneck.

The geometric mean of all benchmarks is 16.26s for no optimizations, 11.43s for **QoQ**, 2.86s for **Dynamic** sync-coalescing, 2.39s for **Static** sync-coalescing, 1.68s with all optimizations, and 1.31 with all optimizations and using TCMalloc.

The net effect is that the final SCOOP/Qs runtime is ~12× faster than the basic runtime. A comparison of all geometric means for concurrent and parallel workloads is given in Table 6.3

Task	None	Dyn.	Static	QoQ	All	All/TC
Parallel	49.14	1.90	1.35	38.91	1.33	1.32
Concurrent	5.38	4.30	4.24	3.36	2.11	1.31
Both	16.26	2.86	2.39	11.43	1.68	1.31

Table 6.3: Geometric means (in seconds) benchmarks with optimizations

6.3 Implementation comparison

The Qs execution approach is validated with two separate implementations, the SCOOP/Qs implementation and the EVE/Qs implementation. The SCOOP/Qs implementation has the benefit of being a fresh start, and represents what is possible when the entire compilation and run-time approach can be tailored exactly to meet SCOOP's needs. The EVE/Qs implementation does not have this luxury, and must be compatible with the existing EiffelStudio run-time and compilation approach.

6.3.1 Parallel

As explained in Section 5.4.4, the EVE/Qs run-time shares the queue-of-queues model with SCOOP/Qs, as well as the dynamic avoidance of sync operations. Section 6.2 shows that dynamic removal of sync operations greatly reduces the execution time of the parallel benchmarks, and lowers the communication overhead significantly.

Execution time. The effect of sync operation removal can also be seen in the reduction in total execution time for the EVE/Qs implementation, shown in Figure 6.4. As with the comparison of individual optimizations, chain does not

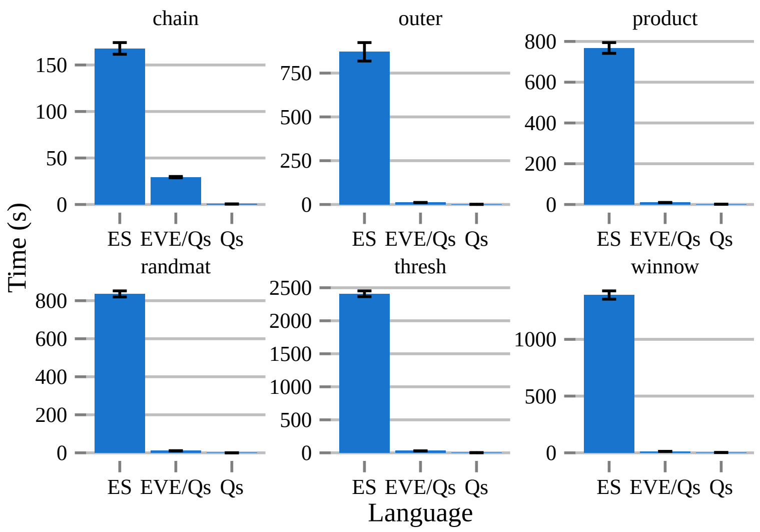


Figure 6.4: Parallel comparison at 32 cores with EiffelStudio, EVE/Qs and Qs

have the same magnitude of improvement because it has less communication overhead than the other parallel benchmarks. The performance of EVE/Qs is about an order of magnitude slower than SCOOP/Qs, mostly due to the compilation and execution approach of Eiffel programs (not just SCOOP) as done by the EiffelStudio compiler and run-time. In particular, the garbage collection scheme used in EiffelStudio requires that the address of the traceable stack variables (such as arrays) are stored in a secondary stack. This secondary stack will be used during the marking phase of garbage collection. The reason this is a problem is because taking the address of a stack variable limits what the underlying optimizer can do because suddenly those variables could be modified out of view of the current function that the optimizer is working on, and the optimizer has to assume that bad things do happen. This is especially important when there's a tight loop that works on an array of integer values, as in the Cowichan problems, because the optimizer can often do a lot to improve the performance through vectorization. Because of this, it is not seen as essential to implement the **Static** sync removal optimization, as the optimizer wouldn't be able to use this information effectively in any case.

Multicore scaling. In addition to the low absolute performance, the scaling characteristics of EiffelStudio's SCOOP implementation are also not what would be expected. In particular, the single-core performance is constrained. This can be seen in Figure 6.5 in the jump from 2-core execution to 4-core execution in the

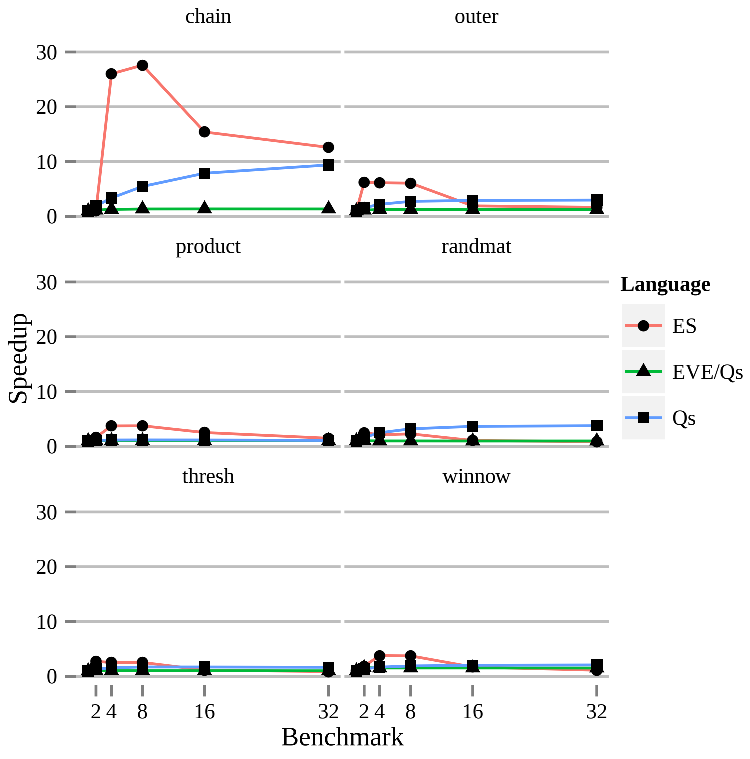


Figure 6.5: Parallel speedup comparison with EiffelStudio, EVE/Qs and Qs

chain benchmark. This increase should be at most double, but for EiffelStudio it is over 25x greater. This cannot be due to the additional resources allocated to perform the computation of the benchmark, even allowing for cache effects. Rather it must be that the additional computational resources help overcome the inherent inefficiency in EiffelStudio's SCOOP runtime.

Profiling reveals that most of the time is spent in the loop that applies calls and notifies clients. Also `mutrace` [85], a mutex and condition variable profiling tool, reports that when run under 2 cores, the time spent waiting on condition variables in the chain benchmark is about 75.0s [1]. When running the same benchmark with 3 cores, the time drops by an order of magnitude, down to 6.5s spent waiting on condition variables. This data suggests, although it is not conclusive, that since the handler is running more slowly the clients are context switching more as they wait for the query result to arrive. When the number of cores is increased, there are more resources for the clients and handler, thus the handler can more readily process incoming calls and the clients do not context switch as often as a result.

As the number of cores continue to increase, the performance degrades again (in all parallel benchmarks) for the EiffelStudio implementation. It is not clear why this occurs, although it is possible that the effect is the same as with the **None** optimization and there is a lot of context switching. However the effect here is much more pronounced that in the **None** case. These effects are not seen in the EVE/Qs or SCOOP/Qs implementations as they both have some form of sync reduction mechanism. Table 6.4 displays the times for each approach for each number of cores used.

Task	Impl.	1	2	4	8	16	32
chain	ES	2088.56 [2]	2088.56	80.28	75.69	135.65	165.84
chain	EVE/Qs	40.11	35.01	31.51	29.75	29.56	29.62
chain	Qs	5.36	2.86	1.60	0.98	0.68	0.57
outer	ES	1424.82	229.49	232.13	235.19	743.51	865.28
outer	EVE/Qs	13.92	12.24	11.18	11.21	11.32	11.39
outer	Qs	2.55	1.62	1.16	0.93	0.87	0.86
product	ES	1148.16	694.85	307.99	306.02	455.02	777.05
product	EVE/Qs	10.00	9.73	9.74	9.75	9.76	9.77
product	Qs	1.44	1.31	1.25	1.23	1.24	1.31
randmat	ES	718.37	287.85	341.30	315.25	675.46	838.30
randmat	EVE/Qs	10.60	10.45	10.78	10.77	10.88	10.84
randmat	Qs	0.72	0.44	0.29	0.22	0.20	0.19
thresh	ES	2092.66	764.67	829.01	827.09	1837.08	2405.55
thresh	EVE/Qs	30.57	29.93	29.73	29.72	29.96	30.13
thresh	Qs	3.57	2.71	2.28	2.09	2.10	2.16
winnow	ES	1557.49	830.03	411.66	416.53	917.25	1392.29
winnow	EVE/Qs	19.47	12.33	12.72	12.74	12.74	12.81
winnow	Qs	5.09	3.71	3.03	2.68	2.52	2.45

Table 6.4: Precise times (in seconds) for parallel benchmark of EiffelStudio, EVE/Qs, and SCOOP/Qs

The geometric means of times for all parallel benchmarks give an average

[1] This profiling was not done on the benchmarking machine, which was difficult to get mutrace working on.

[2] This test would not run to completion so the time is copied from the 2-core case.

Task	EiffelStudio	EVE/Qs	SCOOP/Qs
chameneos	97.55	51.51	4.19
condition	205.88	35.60	1.46
mutex	2.31	0.90	0.14
prodcons	120.64	25.62	0.88
threadring	468.71	112.19	5.08

Table 6.5: Precise times (in seconds) for concurrency benchmark of EiffelStudio, EVE/Qs, and SCOOP/Qs

runtime of 580.54s for EiffelStudio, 15.94s for EVE/Qs, and 1.32s for SCOOP/Qs. There is at least an order of magnitude between each of the implementations.

6.3.2 Concurrent

Unlike the parallel benchmarks, the concurrent benchmarks do not contain performance irregularities. Additionally, Figure 6.6 shows that the performance of EiffelStudio and EVE/Qs is much closer in the concurrent benchmarks. EVE/Qs

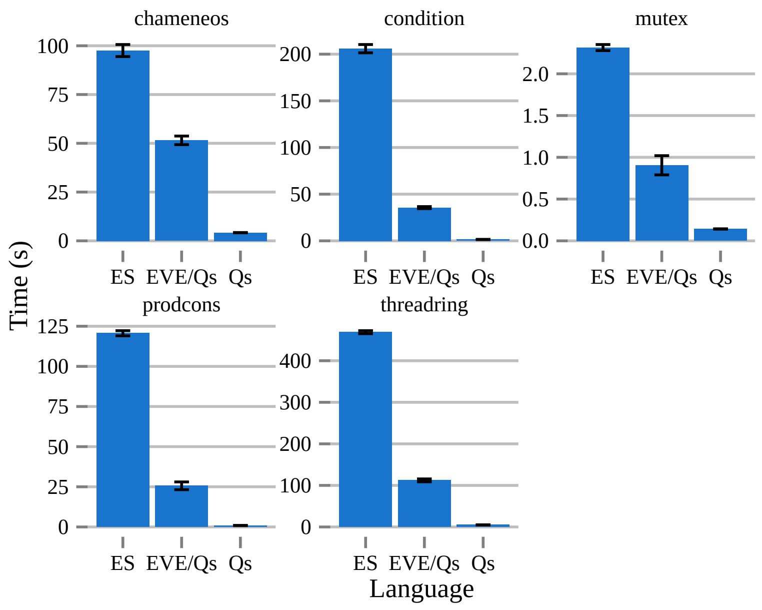


Figure 6.6: Concurrent comparison with EiffelStudio, EVE/Qs and Qs

is still faster, in all cases, than EiffelStudio, although not by nearly as much as with the parallel benchmarks.

This indicates the EiffelStudio run-time is more tailored to programs that follow a concurrent pattern and do not transmit as much data between handlers, as in the parallel benchmarks. Table 6.5 shows the precise times of the benchmarks. The geometric means, again, tell the general story: the difference between EiffelStudio and EVE/Qs is no longer an order of magnitude, but a constant of about 3x. Concretely, the geometric mean for EiffelStudio is 75.54s, EVE/Qs has 21.66s, and Qs takes 1.31s. The order of magnitude difference between Qs and

EVE/Qs remains, again due the fundamental inefficiencies in the non-SCOOP part of the EiffelStudio run-time, which EVE/Qs also uses. Section 5.4.4 shows that one of these is the use of locks in all allocations in EiffelStudio's memory allocator, which is even used to allocate shadow-stack frames for the garbage collection at the start of every routine. SCOOP/Qs also uses lightweight threads which reduces the thread-switching overhead, which is also a factor in both the concurrent and parallel benchmarks.

6.3.3 Summary

The geometric means of each benchmark set for each language are given in Table 6.6. The geometric means of both problem sets is also given, from which

Task	EiffelStudio	EVE/Qs	SCOOP/Qs
Parallel	580.54	15.94	1.32
Concurrent	75.54	21.66	1.31
Both	209.41	18.58	1.31

Table 6.6: Geometric means of times (in seconds) for EiffelStudio, EVE/Qs, SCOOP/Qs in different benchmark sets, and combined.

one can see that overall trend continues: EVE/Qs is an order of magnitude faster than EiffelStudio's SCOOP implementation. Additionally SCOOP/Qs is still an order of magnitude faster than EVE/Qs, and two orders of magnitude faster than EiffelStudio.

6.4 Language comparison

Comparing languages controls the fewest number of variables: the optimizations are different, the implementations are different, and the semantics of the language vary widely. The only thing that remains the same are the benchmarks that are tested. The purpose of such a comparison is to understand the different trade-offs that come with language choices.

The comparison languages should: be currently used, well-known, have mature implementations, and represent a variety of different underlying design choices. For the purposes of the evaluation, this means that the evaluation should select from different programming paradigms, different approaches to shared memory, different concurrency safety guarantees, and different threading implementations. With this in mind, the selected languages are: C++/TBB (Threading Building Blocks) [51], Erlang [4], Go [35], and Haskell [82]. This selection attempts to combine a reasonable number of the facets outlined above to give a complete picture.

C++/TBB. The inclusion of C++ represents the kind of "traditional" approach to concurrency that uses regular threads and shared memory. TBB is a library containing many functions, in the context of these benchmarks it is used to aid the implementation of the parallel benchmarks. In particular TBB contains routines that map over an array in parallel, and also can reduce an array in parallel given an associative operator. TBB is not used for the concurrency benchmarks, as those are either taken from the Shootout Game's repository or use simpler primitives like mutexes and condition variables.

Go. Go is a recent language from Google, modelling its style of concurrency after CSP by including channels for communication between threads. However, Go still allows the usage of shared memory and is thus still susceptible to data races. Go is differentiated from C++/TBB in that it uses light-weight threads as opposed to operating system threads.

Erlang. Erlang is a functional language developed by Ericsson that is designed for highly concurrent systems. It disallows data races by having isolated processes that communicate purely through sending and receiving messages to each other. Erlang has no sharing between different processes: when data is sent between processes it is copied in its entirety. Additionally, it uses light-weight threads as the underlying threading mechanism.

Haskell. As purely functional language, Haskell requires that actions that may change the state be indicated in the type of the action. In particular this allows approaches like STM to be used in a safe way, by mandating that transactional variables are never modified outside of an atomic region. This property is used in the implementations of the concurrency benchmarks programs to provide data race freedom. The parallel benchmarks use the Repa library and the `par` function to provide data race freedom. Repa allows high-level descriptions of array and matrix operations to be executed safely in parallel. The `par` function operates as an advisory to the Haskell run-time that its arguments may be evaluated in parallel; `par` cannot introduce data races because it is a pure function.

Table 6.7 makes the diversity of properties clear. The Races column says if

Language	Races	Threads	Paradigm	Memory	Approach
C++/TBB	possible	OS	Imperative	Shared	Skeletons/threads
Go	possible	light	Imperative	Shared	Goroutines/Channels
Haskell	none	light	Functional	STM	STM/Repa
Erlang	none	light	Functional	Non-shared	Actors
SCOOP/Qs	none	light	O-O	Non-shared	Active Objects

Table 6.7: Language characteristics

races are possible, Threads reports what kind of threading is used in the implementation, Paradigm describes the language paradigm, Memory refers to how memory is shared between threads, and Approach is the underlying approach to concurrency that the language uses and what is used in the construction of the benchmark programs.

Note that the implementations of the C++, Go, and Erlang parallel benchmarks were reviewed by external experts. This lessens the risk that some error was made in the implementations by not knowing enough about the paradigm to be effective. The implementations, along with usability and performance comparisons can be found in [76, 75].

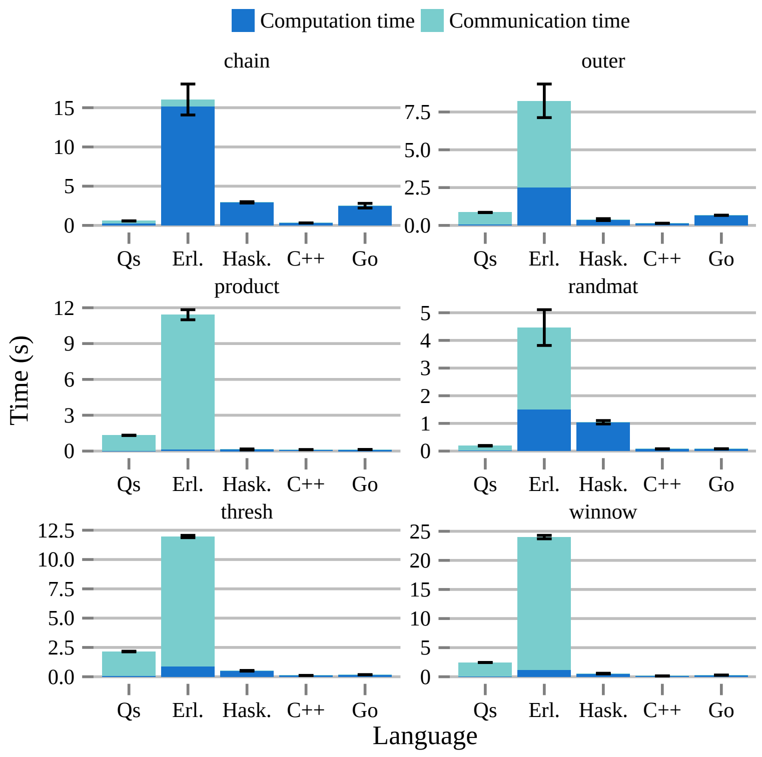


Figure 6.7: Execution times of parallel tasks on different languages, executed on 32 cores

6.4.1 Parallel benchmarks

The parallel benchmarks are meant to measure how well a language can handle taking a particular program and scaling it given more computational resources (cores). Note that it is common in the Erlang and SCOOP/Qs implementations of the Cowichan problems that a significant amount of time is spent sharing results among the threads. Therefore, to more clearly see the effect of different optimizations, and to separate computational effects from communication effects, the time spent computing versus the time spent communicating the results is distinguished.

Execution time. The graph of performance with 32 cores is given in Figure 6.7. As with SCOOP/Qs, to give a more clear picture of the performance characteristics of Erlang, the computation and the communication time are also distinguished here. One can see that SCOOP/Qs and Erlang both spend a majority of their time in communication, with the exception of the chain problem, which has much less communication between the workers. It is useful to consider both the total and the computation time: in non-benchmark style problems it is more likely that the workloads fall somewhere in the middle. For example, the chain problem, which is the composition of the other smaller benchmarks, does not suffer from nearly the same communication burden as they do.

Erlang has unfavourable performance results compared to the other lan-

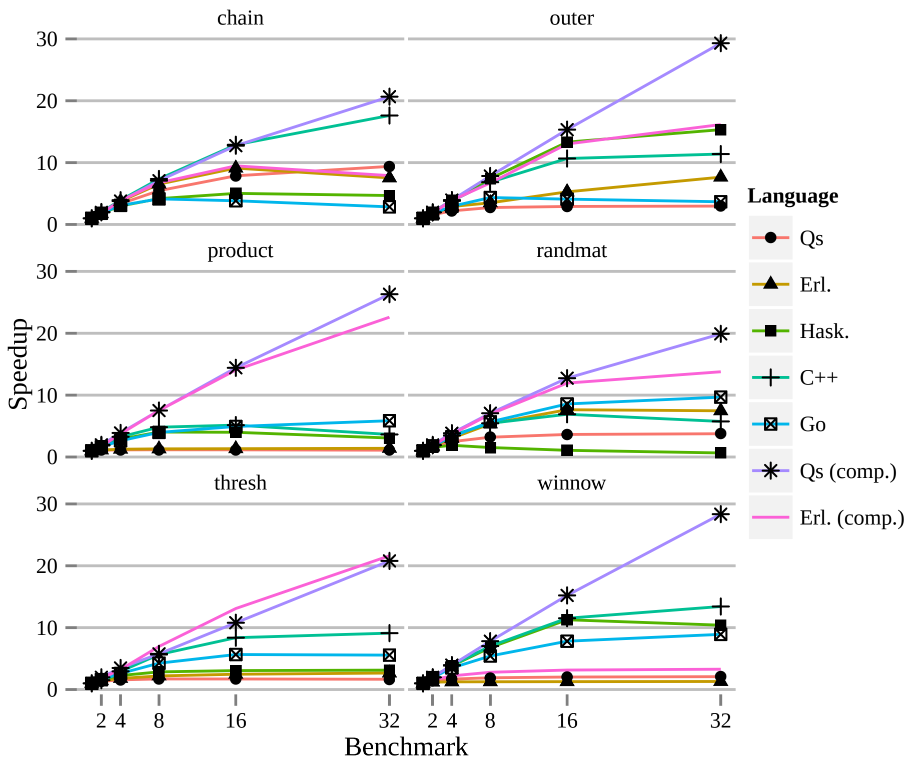


Figure 6.8: Speedup over single-core performance, up to 32 cores

guages. Due to Erlang's data representation (forces to use linked lists to represent matrices) and its execution model (cannot use the HiPE-optimized builds [41] with the multithreaded run-time), it generally falls far behind the other approaches, even Haskell and SCOOP/Qs.

Besides Erlang, the other languages are more closely grouped. The geometric means for total time are, in increasing order: C++/TBB (0.32s), Go (0.57s), Haskell (0.89s), SCOOP/Qs (1.32s), and Erlang (18.07s). For computation-only time, the order is: SCOOP/Qs (0.28s), C++/TBB (0.32s), Go (0.57s), Haskell (0.89s) and Erlang (4.32s).

Note that this puts SCOOP/Qs first because many of the cache effects are removed due to the predistribution of the data before the timing starts. This is only included as a sanity test, to show that the lower-bound for the SCOOP/Qs implementation is competitive with the other approaches.

Multicore scaling. The other aspect that is investigated is the speedup of the benchmarks across 32 cores. In Figure 6.8 one can see the performance of the various languages on the different problems. One can additionally see that on chain, most languages manage to achieve a speedup of around 5x. Go is the exception, and runs into problems past 8 cores. This was also an effect that was seen in a previous language comparison study [76] from which the implementation was taken; the implementation was also reviewed by a key Go developer in the study. Erlang also sees a performance degradation, though only

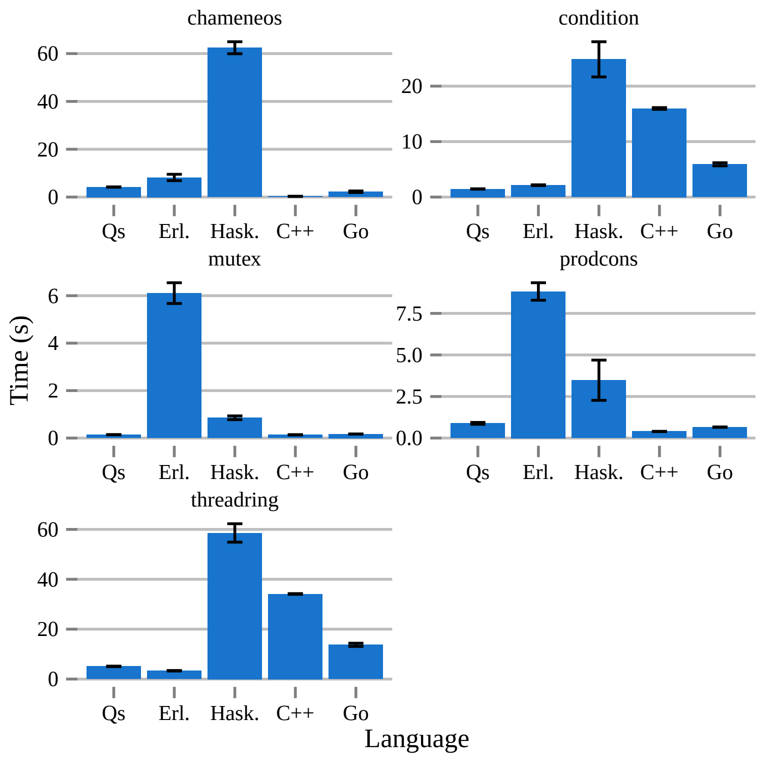


Figure 6.9: Execution times of concurrent tasks on different languages

past 16 cores.

Also of note is the performance of Haskell on the randmat benchmark. This is one of the few benchmarks where Repa could not be effectively used due to the nature of the problem, so the basic par-based concurrency primitives are used. The basic strategy has chunks of the output array constructed in parallel, then concatenated together. The concatenation is sequential, however, putting a limit on the maximum speedup; using the ThreadScope [108] performance reporting tool, it was clear that the stop-the-world garbage collector was intervening too often. The last unexpected result was the inability for the Erlang version of the winnow program to speedup past about 2-3x. This was examined in detail but no cause could be found. Precise timing data can be found in Table 6.8.

6.4.2 Concurrent benchmarks

The concurrent programming tasks are compared in Figure 6.9 and exact times in Table 6.9.

Haskell tends to perform the worst, which is likely due to the use of STM, which incurs an extra level of bookkeeping on every operation. Erlang performs better, but in general lags behind the other approaches. C++/TBB tends to be the fastest, except in the condition and threadring benchmarks, which are both essentially single-threaded; they are designed to test context switching overhead in various forms. Go does quite well uniformly, never the fastest, but never the slowest. Lastly, SCOOP/Qs performs mostly in line with C++/TBB and Go,

		Threads					
Task	Language	1	2	4	8	16	32
chain	Qs	5.36	2.86	1.60	0.98	0.68	0.57
chain	Erlang	120.59	69.00	32.06	18.48	13.23	16.01
chain	Haskell	13.78	7.71	4.62	3.30	2.74	2.94
chain	C++	5.57	2.76	1.42	0.76	0.43	0.32
chain	Go	7.39	4.09	2.39	1.79	1.93	2.60
chain	Qs (comp.)	5.30	2.70	1.40	0.75	0.42	0.26
chain	Erlang (comp.)	119.68	68.13	30.93	17.75	12.63	15.15
outer	Qs	2.55	1.62	1.16	0.93	0.87	0.86
outer	Erlang	61.57	38.21	21.19	17.57	11.67	8.05
outer	Haskell	5.49	2.76	1.40	0.74	0.41	0.36
outer	C++	1.59	0.83	0.42	0.23	0.15	0.14
outer	Go	2.47	1.44	0.84	0.57	0.60	0.67
outer	Qs (comp.)	1.84	0.92	0.47	0.23	0.12	0.06
outer	Erlang (comp.)	40.66	22.54	10.45	6.05	3.12	2.52
product	Qs	1.44	1.31	1.25	1.23	1.24	1.31
product	Erlang	15.89	13.94	12.66	12.08	11.82	11.33
product	Haskell	0.45	0.25	0.16	0.11	0.11	0.15
product	C++	0.44	0.23	0.13	0.09	0.08	0.12
product	Go	0.76	0.46	0.29	0.19	0.15	0.13
product	Qs (comp.)	0.28	0.14	0.07	0.04	0.02	0.01
product	Erlang (comp.)	3.35	1.95	0.90	0.45	0.24	0.15
randmat	Qs	0.72	0.44	0.29	0.22	0.20	0.19
randmat	Erlang	30.93	18.01	10.20	5.77	4.05	4.14
randmat	Haskell	0.68	0.43	0.36	0.44	0.62	1.03
randmat	C++	0.44	0.23	0.13	0.08	0.06	0.08
randmat	Go	0.78	0.43	0.24	0.14	0.09	0.08
randmat	Qs (comp.)	0.58	0.30	0.15	0.08	0.05	0.03
randmat	Erlang (comp.)	20.69	11.26	5.63	2.99	1.73	1.50
thresh	Qs	3.57	2.71	2.28	2.09	2.10	2.16
thresh	Erlang	31.82	22.35	17.77	14.48	12.88	11.96
thresh	Haskell	1.56	0.96	0.69	0.55	0.51	0.50
thresh	C++	1.00	0.66	0.34	0.18	0.12	0.11
thresh	Go	0.95	0.60	0.37	0.22	0.17	0.17
thresh	Qs (comp.)	1.76	0.89	0.51	0.31	0.16	0.08
thresh	Erlang (comp.)	19.30	10.74	5.97	2.77	1.47	0.89
winnow	Qs	5.09	3.71	3.03	2.68	2.52	2.45
winnow	Erlang	31.03	26.02	25.04	24.75	24.38	23.95
winnow	Haskell	5.43	2.77	1.42	0.80	0.48	0.52
winnow	C++	2.04	1.03	0.53	0.29	0.18	0.15
winnow	Go	2.47	1.29	0.71	0.46	0.32	0.28
winnow	Qs (comp.)	2.76	1.40	0.70	0.35	0.18	0.10
winnow	Erlang (comp.)	4.06	2.58	1.84	1.46	1.29	1.24

Table 6.8: Parallel benchmark times (in seconds)

Task	SCOOP/Qs	Erlang	Haskell	C++	Go
chameneos	4.19	8.20	62.45	0.32	2.23
condition	1.46	2.14	24.83	15.97	5.91
mutex	0.14	6.11	0.86	0.14	0.17
prodcons	0.88	8.82	3.48	0.40	0.66
threadring	5.08	3.33	58.58	34.12	13.73

Table 6.9: Concurrent benchmark times (in seconds)

however it is the fastest in the condition benchmark. In increasing order of geometric means: SCOOP/Qs (1.31s), C++/TBB (1.57s), Go (1.82s), Erlang (5.01s), and Haskell (12.20s).

6.4.3 Summary

This evaluation presents a wide variety of approaches to concurrency and situates SCOOP/Qs among them. In particular, SCOOP/Qs is the fastest language over all problems on the concurrency benchmarks, beating even C++. Note, however, that neither Go nor C++/TBB offers any of the guarantees of SCOOP/Qs, and SCOOP/Qs offers more guarantees than Erlang, and roughly the same guarantees as Haskell (through transactional memory).

For parallel problems, SCOOP/Qs ranks 4th owning mostly to the communication overhead of the language. The chain benchmark indicates the performance of computational tasks that are not dominated by communication, and SCOOP/Qs is only beat by C++ on that benchmark. SCOOP/Qs outpaces Erlang in all cases, and can provide reasonable performance comparable with the other approaches, while still providing more guarantees in the execution model.

Table 6.10 shows the geometric means for each benchmark set and the combined mean for all problems per language.

Benchmark	SCOOP/Qs	Erlang	Haskell	C++	Go
Parallel	1.32	18.07	0.89	0.32	0.57
Concurrent	1.31	5.01	12.20	1.57	1.82
Both	1.31	9.51	3.30	0.71	1.02

Table 6.10: Geometric means of times (in seconds) for languages in different benchmark sets, and combined.

6.5 Related work

Blackburn et al. [8] provide a comprehensive analysis of benchmark construction, and the associated pitfalls, while proposing the DaCapo benchmark suite as an improvement over the existing SPEC Java benchmark. However, much of the advice is specific to environments which have a JIT compiler or a runtime, and thus more specific than this work.

In [87], Richards et al. develop an automated technique for benchmark construction. It collects traces of Javascript executions from existing websites to model relevant real world behaviour. These traces are used to generate programs representing the traces, which can mimic the non-functional characteristics of the original JavaScript program. This work is specific to the JavaScript language, does not consider concurrent programs, and highlights workloads that are important for JavaScript run-times. The goal of having a more structured benchmark process, although not mechanized, is also sought after in this thesis,

Kalibera et al. [46] propose that concurrent benchmarks should be measure other things than just runtime. To better understand what benchmarks are actually testing the performance of, they record read, write, memory allocation, and monitor acquisition operations. This enables them to examine details such as how much contended locking occurs and how much cache invalidation may happen, although they do not examine the hardware effects directly. They then analyze operation patterns under different time periods and situations, and are able to effectively perform a post-mortem analysis of the DaCapo benchmark suite's concurrent application workloads. Such fine grained measurements would also be interesting in the different comparisons between optimizations, implementations, and languages.

Guerraoui et al. [38] also aim to take a look at benchmarking software transactional memory (STM) implementations. Like [46] it bases much of its analysis not in specific benchmarks but rather focuses on shared read/write behaviour. It proposes the use of data graphs for STM implementations to traverse and modify in a variety of ways. These different graphs and workloads then exercise various data access patterns (read-heavy, write-heavy, read/write). This approach bears a slight resemblance to ours in the usage of a variety of structured benchmark workloads, but they are not composable, and they technique is specific to testing STM implementations.

A performance bug study by Jin et al. [44] randomly surveyed 109 performance bugs from open source programs. Of the 109 bugs, 31 were introduced by developers not being aware of the performance implications of a feature or API. One example of this was a lock being taken inside a library function, this performance bug would not be visible in single-threaded execution. Findings such as this suggest the need for a way to re-examine existing benchmark practices, in particular to start combining benchmarks in situations that were not originally considered, i.e. running operations in parallel.

Zhang and Seltzer [112] propose a vector of microbenchmarks for primitive operations (calls, arithmetic, etc.). This vector is then combined (dot-product) with a vector characterizing a particular application's usage of these primitive operations. This separation of measurement and importance is interesting and may provide a more structured way to evaluate benchmarks in particular contexts where different performance characteristics are required.

A wide array of tests on multicore Java performance characteristics were conducted by Sartor and Eeckhout [89]. The experiments were of a wide and interesting variety, examining the effect of frequency scaling and thread migrations on a variety of benchmarks from the DaCapo benchmark suite.

In [99] Turon proposes a new concurrent programming method, and reinforces the validity of the method with benchmarks. Four different benchmarks are used, mostly based on well known data structures, and each test exists in two modes: high and low contention. The goal of providing different workloads is similar to the two categories of benchmarks that are used to evaluate the SCOOP/Qs work.

6.6 Conclusion

This evaluation has shown the performance properties of the SCOOP/Qs execution model. Beginning with the most specific, optimizations that are specific SCOOP/Qs are compared to gauge their effectiveness on different workloads. This has shown that the two main optimizations sync reduction and the queue-of-queues model for handlers both contribute to increasing SCOOP performance. These improve performance by 12× on average over the basic implementation.

Secondly, the same ideas are used in the implementation of the Qs run-time inside the EVE research branch of EiffelStudio to show that they are effective even outside of the ideal SCOOP/Qs implementation. They are shown to increase the performance over the EiffelStudio implementation by an order of magnitude, and the SCOOP/Qs implementation an order of magnitude over EVE/Qs.

Finally comparing SCOOP/Qs against other well known languages/paradigms with well-tested implementations shows the overall suitability of the model. SCOOP/Qs is the fastest on the concurrency benchmarks, and overall is the fastest from the languages that offer data race freedom.

The entire evaluation provides a deep analysis of the performance characteristics of the SCOOP/Qs ideas in a broad array of situations, and comparing many different aspects.

Chapter 7

Conclusion

Throughout this thesis, there has been a focus on providing both correctness and execution efficiency for concurrent object-oriented programs. This is shown by the research advances that work to improve the efficiency and correctness in a concurrent programming model.

The common thread underlying the work is the object-oriented concurrent programming language, SCOOP. This language forms a structure upon which many of the important advances are made, and allows the different contributions to be compared and contrasted under within a single frame of reference. Its disciplined approach to concurrent programming also enables the techniques that comprise this work.

7.1 Deadlock

The structured approach to ensuring mutual exclusion through the reservation of separate handlers is one example. Having built in structure enables the creation of a type system which eliminates the occurrence of deadlock due to the reservation of handlers. The type system extension that the work proposes allows the programmer to be free of deadlocks and can even be augmented with simple inference mechanisms to lessen the annotation burden. The benefits of the system are shown by applying it to the example of a web server, and demonstrates an annotation overhead on this example of of 4%, dropping to 0.5% when the lock set and order inference algorithm is used. The quality of the system is reinforced by a mechanized proof of soundness in the Coq proof assistant. These properties provide evidence that the technique is both usable and correct.

7.2 Concurrent Testing

There are many properties, though, that are not as easily expressed in the type system, at least not without significant effort on the part of the programmer. For these cases, testing is a natural solution, as it provides a method to highlight, specify and verify important functional properties. In the context of SCOOP, a language that contains built-in support for contracts, this leads towards a technique that can take advantage of such program specifications. The demonic testing work combines specifications with the dynamic state of the program during unit testing to create a testing method that can test concurrent programs in a concurrent setting while maintaining determinism and modularity. In particular, demonic testing can detect the existence of data races (if the language allows for them) and also atomicity violations in the case of SCOOP. Demonic testing's reasoning engine, demonL, allows interference to be generated on the fly, representing the errors that could be caused by other threads in the system. demonL is built on top of the abilities provided SMT solvers. This technique is applied bugs from well known, large, applications such as Apache and Firefox, and is found to offer good detection abilities and testing times.

7.3 Execution

Deadlock and atomicity violations address termination and functional correctness issues that may arise in SCOOP programs. One problem that does *not* arise in SCOOP are data races, but ensuring that data races do not occur incurs a run-time cost. The SCOOP/Qs work analyzes the existing guarantees that SCOOP provides and proposes an alternative execution model that, while still providing these guarantees, can be implemented more efficiently and thus addresses one of the major reasons for constructing a concurrent program: speed. The SCOOP/Qs semantics also reduce the possibility for deadlock by proposing a handler reservation mechanism that does not block. Modifications to the execution model are carried through to the run-time and compilation process, proposing implementation-level optimizations as well as a compiler optimization pass, resulting in a fresh implementation of the SCOOP language. The techniques are also incorporated into the EVE project, based on the commercial EiffelStudio IDE.

7.4 Evaluation

The SCOOP/Qs model is shown to be highly effective at reducing the execution time required for SCOOP programs, improving the performance one and two orders of magnitude as more SCOOP/Qs optimizations are employed. A comprehensive evaluation shows where each optimization technique demonstrates its strength, with the major optimizations being sync reduction and the queue-of-queues approach to call logging together providing an order of magnitude speedup over a basic implementation. The different approaches taken in different SCOOP implementations is also compared, showing that EVE/Qs and SCOOP/Qs are one and two orders of magnitude faster than the current EiffelStudio implementation. Lastly, SCOOP/Qs is compared against other well known languages that all employ different execution approaches (C++, Go, Erlang, Haskell). SCOOP/Qs is found to be the fastest on the concurrent benchmarks out of all languages, and fastest on average among the race-free languages.

7.5 Summary

This thesis addresses the major concerns of correctness and execution efficiency, and shows how to attain enable greater assurance of functional correctness and faster execution speed. These aspects are targeted to provide a comprehensive approach to improving the state concurrent programming.

7.6 Future work

Deadlock. Although the deadlock freedom scheme presented in Chapter 3 offers features which lessen the annotation burden, this could be taken further. In particular, the technique could try to use a Damas-Milner [26] type reconstruction approach to also infer the generic handlers of routines and classes, which currently must be given manually.

Additionally, the proof of the core soundness guarantees covers an important but incomplete part of the SCOOP semantics. This could be extended to model the full SCOOP semantics as opposed to the more limited operation semantics that is in the proof currently. A full proof of the property would provide a greater degree of assurance of soundness for the technique. Filling out the semantic formalization in Coq, based on the existing proof, would also provide a useful starting point for other proofs of execution guarantees that SCOOP aims to provide, such as data race freedom and fairness of some operations.

Lastly, the overall approach to guaranteeing deadlock freedom could be refined to take into account the changes in the SCOOP/Qs model, in particular that locking no longer happens when a call is made with separate arguments. Since the only unbounded blocking calls are queries in SCOOP/Qs, the new technique would only have to worry about locking order when queries are made; only when a query is made would there be a waits-for edge in the locking graph.

Concurrent Testing. The approach to testing concurrent programs in Chapter 4 replaces the actual interference from external threads with "logical interference" via a logical description of what interference could arise. However, the effectiveness of this relies on the quality of the rely condition, and also on the precision of the postconditions of routines in the domain. False positives result from inaccurate rely-conditions and loose postconditions, thus a secondary filter could be implemented that takes the proposed interference generated by demonL and actually runs the sequence to determine if it is a real error. This could be done in an independent run to the testing run so that if a false positive is detected it would not affect other tests.

Although drawing on the ideas of rely-guarantee reasoning, the demonic testing process currently does not support stating or checking guarantee conditions. Guarantee checking would add another degree of safety and would come in two parts

1. Checking the guarantees are respected by those that state them; checking would be limited to a run-time prestate/poststate after every instruction.

2. Verifying (statically) that the rely and guarantee specifications of a particular thread consistent. Take two threads executing in parallel

$$\{P, R\} t_1 || t_2 \{G, Q\}$$

where R and G are global rely and guarantee specifications, and the per-thread rely and guarantee specifications are (R_1, G_1) and (R_2, G_2). For these to composes properly in parallel, these specifications must satisfy:

$$R \vee G_2 \Rightarrow R_1 \wedge R \vee G_1 \Rightarrow R_2 \wedge G_1 \wedge G_2 \Rightarrow G$$

Execution. The SCOOP/Qs execution model given in Chapter 5 consists of three parts, an operational semantics, a compiler with specific optimizations, and an efficient run-time implementation. However, as a prototype it is always in need of more and larger examples to show its suitability and also its rough edges. Work has already begun on a SCOOP/Qs HTTP server, which is showing promise but to fulfill that promise should be further developed.

The recent inclusion of an IO manager (a thread which schedules IO operations for lightweight threads), now adds the possibility to implement distributed `separate` objects. This would substantiate the SCOOP/Qs design that uses independent interaction between handlers through private queues, as this strongly resembles communication over a network. However, not all existing optimizations would be valid, such as client-side query execution which assumes a shared memory space, and would have to be re-examined.

Additionally, there is currently no garbage collection system for SCOOP/Qs, something that would be useful to have, and the SCOOP/Qs run-time provides an efficient starting point from which to construct a concurrent garbage collector specific to the SCOOP model.

Lastly, given the showing in the parallel benchmarks, it is clear that SCOOP is not strongly suited for these workloads. SCOOP/Qs, as the most efficient SCOOP implementation, is therefore a natural place to investigate techniques to allow for safe shared-memory computation by analyzing and changing the semantic model of the SCOOP language. One way would be to introduce a way for handlers to give ownership of some of their data to another handler to process. Such an approach would require adding a linear type system

Benchmarking. Chapter 6 shows a broad array of measurements of SCOOP/Qs and other languages and implementations. To make SCOOP more readily comparable to other languages, it would be useful to expand the number and variety of benchmarks. These could include more of the concurrent/parallel benchmarks from [24], or other standard benchmark algorithms such as fast Fourier transform and successive over-relaxation.

Internally, the benchmarks in the evaluation are used to discover workloads that are slower than other languages and prompt an examination of why. The task of finding why something is slow, however, is not always simple. It requires debugging performance using standard tools which are not aware of the run-time structure of SCOOP/Qs programs. Therefore, as a way to better explain performance results, it would be useful to have a profiling technique specific to the SCOOP/Qs run-time much as was done in [71], but also at a lower level where the level of abstraction is in terms of run-time, not SCOOP, operations.

Bibliography

[1] Andrei Alexandrescu. *The D Programming Language*. Addison-Wesley, 1st edition, 2010.

[2] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[3] Wladimir Araujo, Lionel Briand, and Yvan Labiche. On the effectiveness of contracts as test oracles in the detection and diagnosis of race conditions and deadlocks in concurrent object-oriented software. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 2011.

[4] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in Erlang*. Prentice Hall, Hertfordshire, UK, 2nd edition, 1996.

[5] Mike Barnett, Boryuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the International Symposium on Formal Methods for Components and Objects*, number 4111 in Lecture Notes in Computer Science, pages 364–387. Springer, 2005.

[6] Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI C Specification Language (V1.8)*, March 2014.

[7] Saddek Bensalem, Jean-Claude Fernandez, Klaus Havelund, and Laurent Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pages 41–50. ACM, 2006.

[8] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot, B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, pages 169–190. ACM Press, 2006.

[9] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1):1636–1642, 1995.

[10] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, New York, NY, USA, 1995. ACM.

[11] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, pages 356–368. IEEE, 1994.

[12] Colin Blundell, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Assume-guarantee testing. In *Proceedings of the Workshop on Specification and Verification of Component-Based Systems*. ACM, 2005.

[13] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, May 1966.

[14] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 211–230. ACM, 2002.

[15] Phillip J. Brooke, Richard F. Paige, and Jeremy L. Jacob. A CSP model of Eiffels SCOOP. *Formal Aspects of Computing*, 19(4):487–512, 2007.

[16] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: The new adventures of old X10. In *Proceedings of the International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 51–61. ACM, 2011.

[17] Chair of Software Engineering, ETH. EVE (Eiffel Verification Environment). `http://se.inf.ethz.ch/research/eve/`, March 2014.

[18] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[19] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59. ACM, 1998.

[20] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. *ACM SIGPLAN Notices*, 33(10):48–64, 1998.

[21] Code contracts. `http://research.microsoft.com/en-us/projects/contracts/`, 2011.

[22] Edward G. Coffman, Michael Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

[23] Collection of Concurrency Bugs. `http://www.eecs.umich.edu/~jieyu/bugs.html`, 2011.

[24] Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/`, 2013.

[25] Patrick R. Conrad, Julie A. Shah, and Brian C. Williams. Flexible execution of plans with choice. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 2009.

[26] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212, New York, NY, USA, 1982. ACM.

[27] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq SRC, 1998.

[28] Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[29] Juergen Dingel. Computer-assisted assume/guarantee reasoning with VeriSoft. In *Proceedings of the International Conference on Software Engineering*, pages 138–148. IEEE Computer Society, 2003.

[30] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the International Conference on Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[31] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[32] EVE project. `https://svn.eiffel.com/eiffelstudio/branches/eth/eve/`, 2011.

[33] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245. ACM, 2002.

[34] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the of ACM Symposium on Parallelism in Algorithms and Architectures*, pages 79–90, New York, NY, USA, 2009. ACM.

[35] Go programming language. `http://golang.org/`, 2013.

[36] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186. ACM, 1997.

[37] Olivier Gruber and Fabienne Boyer. Ownership-based isolation for concurrent actors on multi-core machines. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 7920, pages 281–301. Springer, July 2013.

[38] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: a benchmark for software transactional memory. In *Proceedings of the European Conference on Computer Systems*, pages 315–324. ACM, 2007.

[39] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 1–12. IEEE, 2009.

[40] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular Actor formalism for artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 235–245, San Francisco, CA, USA, 1973.

[41] HiPE reference manual. `http://erlang.org/doc/apps/hipe/`, 2013.

[42] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[43] Bart Jacobs, Jans Smans, Frank Piessens, and Wolfram Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *Proceedings of the International Conference on Formal Engineering Methods*, volume 4260 of *Lecture Notes in Computer Science*, pages 420–439. Springer, 2006.

[44] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 77–88. ACM, 2012.

[45] Cliff B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981.

[46] Tomas Kalibera, Matthew Mole, Richard Jones, and Jan Vitek. A black-box approach to understanding concurrency in DaCapo. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems, Languages, and Applications*. ACM, October 2012.

[47] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the European Conference on Artificial Intelligence*, pages 359–363. Wiley, 1992.

[48] Henry Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 318–325. Morgan Kaufmann, 1999.

[49] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 261–272. ACM, 2010.

[50] Eric Kerfoot, Steve McKeever, and Faraz Torshizi. Deadlock freedom through object ownership. In *Proceedings of the International Workshop on Aliasing Confinement and Ownership in ObjectOriented Programming*, pages 1–8. ACM, 2009.

[51] Wooyoung Kim and Michael Voss. Multicore desktop programming with Intel Threading Building Blocks. *IEEE Software*, 28(1):23–31, 2011.

[52] Naoki Kobayashi. A new type system for deadlock-free processes. In *Proceedings of the International Conference on Concurrency Theory*, pages 233–247. Springer-Verlag, 2006.

[53] Joseph A. Korty. Sema: A lint-like tool for analyzing semaphore usage in a multithreaded UNIX kernel. In *Proceedings of the USENIX Winter Technical Conference*, 1989.

[54] Sudipta Kundu, Malay K. Ganai, and Chao Wang. Contessa: Concurrency testing augmented with symbolic analysis. In *Proceedings of the International Conference on Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 127–131. Springer, 2010.

[55] R. Greg Lavender and Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. In *Pattern languages of program design*, pages 483–499. Addison-Wesley, 1996.

[56] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical report, July 19 1999.

[57] libffi. `http://sourcware.org/libffi/`, March 2014.

[58] LLVM. `http://www.llvm.org`, March 2014.

[59] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[60] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *ACM SIGPLAN Notices*, 43(3):329–339, 2008.

[61] Roberto Lublinerman, Jisheng Zhao, Zoran Budimlić, Swarat Chaudhuri, and Vivek Sarkar. Delegated isolation. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 885–902. ACM, 2011.

[62] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. AutoBlackTest: Automatic black-box testing of interactive applications. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, pages 81–90, 2012.

[63] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960.

[64] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL: The planning domain definition language. Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control, 1998.

[65] Bertrand Meyer. Eiffel: programming for reusability and extendibility. *ACM Sigplan Notices*, 22(2):85–94, 1987.

[66] Bertrand Meyer. Applying “Design by Contract”. *Computer*, 25(10):40–51, October 1992.

[67] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[68] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *IEEE Computer*, 42:46–55, 2009.

[69] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[70] Benjamin Morandi, Sebastian Nanz, and Bertrand Meyer. A formal reference for SCOOP. In *Empirical Software Engineering and Verification*, volume 7007 of *Lecture Notes in Computer Science*, pages 89–157. Springer, 2012.

[71] Benjamin Morandi, Sebastian Nanz, and Bertrand Meyer. Performance analysis of SCOOP programs. *Journal of Systems and Software*, 85(11):2519–2530, 2012.

[72] Benjamin Morandi, Sebastian Nanz, and Bertrand Meyer. Safe and efficient data sharing for message-passing concurrency. In *Proceedings of the International Conference on Coordination Models and Languages*, Lecture Notes in Computer Science. Springer, 2014. To appear.

[73] Benjamin Morandi, Mischael Schill, Sebastian Nanz, and Bertrand Meyer. Prototyping a concurrency model. In *Proceedings of the International Conference on Application of Concurrency to System Design*, pages 177–186. IEEE, 2013.

[74] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280. USENIX Association, 2008.

[75] Sebastian Nanz, Scott West, and Kaue Soares da Silveira. Examining the expert gap in parallel programming. In *Proceedings of the European Conference on Parallel Processing*. Springer, 2013.

[76] Sebastian Nanz, Scott West, Kaue Soares da Silveira, and Bertrand Meyer. Benchmarking usability and performance of multicore languages. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. ACM, 2013.

[77] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. *ACM SIGPLAN Notices*, 42(6):22–31, 2007.

[78] Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, ETH Zurich, 2007.

[79] Jonathan S. Ostroff, Faraz Torshizi, Hai Feng Huang, and Bernd Schoeller. Beyond contracts for concurrency. *Formal Aspects of Computing*, 21(4):319–346, 2009.

[80] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36. ACM, 2009.

[81] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[82] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, New York, NY, USA, 1996. ACM.

[83] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2005.

[84] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[85] Lennart Poettering. mutrace. `http://0pointer.de/blog/projects/mutrace.html`, May 2014.

[86] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[87] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of JavaScript benchmarks. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 677–694. ACM, 2011.

[88] Jussi Rintanen. Heuristics for planning with SAT and expressive action definitions. In *Proceedings of the International Conference on Automated Planning and Scheduling*. AAAI, 2011.

[89] Jennfer B. Sartor and Lieven Eeckhout. Exploring multi-threaded Java application performance on multicore hardware. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 281–296. ACM, 2012.

[90] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[91] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 275–299, Berlin, Heidelberg, 2010. Springer-Verlag.

[92] Mischael Schill, Sebastian Nanz, and Bertrand Meyer. Handling parallelism in a concurrency model. In *Proceedings of the International Conference on Multicore Software Engineering, Performance, and Tools*, volume 8063 of *Lecture Notes in Computer Science*, pages 37–48. Springer, 2013.

[93] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21. ACM, 2008.

[94] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the International Conference on Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006.

[95] Ohad Shacham, Nathan Grasso Bronson, Alex Aiken, Mooly Sagiv, Martin T. Vechev, and Eran Yahav. Testing atomicity of composed concurrent operations. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 51–64, 2011.

[96] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[97] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.

[98] TCMalloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, May 2014.

[99] Aaron Turon. Reagents: expressing and composing fine-grained concurrency. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 157–168. ACM, 2012.

[100] Chao Wang, Mahmoud Said, and Aarti Gupta. Coverage guided systematic concurrency testing. In *Proceedings of the International Conference on Software Engineering*, pages 221–230. ACM, 2011.

[101] Christopher Weber and Daniel Bryce. Planning and acting in incomplete domains. In *Proceedings of the International Conference on Automated Planning and Scheduling*. AAAI, 2011.

[102] Scott West. Demonic test cases. `http://se.inf.ethz.ch/people/west/demonic-cases/`, 2011.

[103] Scott West. Quicksilver, an implementation of the SCOOP/Qs model. `https://github.com/scottgw/quicksilver`, March 2014.

[104] Scott West. SCOOP deadlock tool. `http://github.com/scottgw/scoop-deadlock`, 2014.

[105] Scott West, Sebastian Nanz, and Bertrand Meyer. A modular scheme for deadlock prevention in an object-oriented programming model. In *Proceedings of the International Conference on Formal Engineering Methods*, pages 597–612, 2010.

[106] Scott West, Sebastian Nanz, and Bertrand Meyer. Demonic testing of concurrent programs. In *Proceedings of the International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science. Springer, 2012.

[107] Scott West, Sebastian Nanz, and Bertrand Meyer. Efficient and reasonable object-oriented concurrency. *Computing Research Repository*, abs/1405.7153, 2014.

[108] Kyle B. Wheeler and Douglas Thain. Visualizing massively multithreaded applications with ThreadScope. *Concurrency and Computation: Practice and Experience*, 22(1):45–67, January 2010.

[109] Gregory V. Wilson and R. Bruce Irvin. Assessing and comparing the usability of parallel programming systems. Technical Report CSRI-321, University of Toronto, 1995.

[110] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the International Symposium on Computer Architecture*, pages 325–336. ACM, 2009.

[111] Wei Zhang, Chong Sun, and Shan Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 179–192, 2010.

[112] Xiaolan Zhang and Margo Seltzer. HBench: Java: An application-specific benchmarking framework for Java virtual machines. In *ACM Java Grande*, pages 62–70. ACM Press, 2000.

[113] Jisheng Zhao, Roberto Lublinerman, Zoran Budimlić, Swarat Chaudhuri, and Vivek Sarkar. Isolation for nested task parallelism. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 571–588. ACM, 2013.

Curriculum vitæ

General information

Name:	Scott West
Date of birth:	December 5, 1983
Nationality:	Canadian
Web page:	`http://se.inf.ethz.ch/people/west/`
E-mail:	`scott.west@inf.ethz.ch`

Education

February 2009 - present ETH Zürich
PhD student in the Chair of Software Engineering, supervised by Prof. Dr. Bertrand Meyer.

May 2007 - December 2008 McMaster University
Masters in Computer Science, supervised by Prof. Dr. Wolfram Kahl.

September 2002 - April 2007 McMaster University
Bachelor in Software Engineering and Society.

Publications

To Run What No One Has Run Before: Executing an Intermediate Verification Language, with Nadia Polikarpova, Carlo A. Furia: RV 2013

Benchmarking Usability and Performance of Multicore Languages, with Sebastian Nanz, Kaue Soares Da Silveira, Bertrand Meyer: ESEM 2013.

Examining the Expert Gap in Parallel Programming, with Sebastian Nanz, Kaue Soares Da Silveira: Euro-Par 2013.

Concurrent Object-Oriented Development with Behavioral Design Patterns, with Benjamin Morandi, Sebastian Nanz, Hassan Gomaa: ECSA 2013.

Demonic Testing of Concurrent Programs, with Sebastian Nanz, Bertrand Meyer: ICFEM 2012.

A Modular Scheme for Deadlock Prevention in an Object-Oriented Programming Model, with Sebastian Nanz, Bertrand Meyer: ICFEM 2010.

Deriving Concurrent Control Software from Behavioral Specifications, with Ganesh Ramanathan, Benjamin Morandi, Sebastian Nanz, Bertrand Meyer: IROS 2010.

A Generic Graph Transformation, Visualisation, and Editing Framework in Haskell, with Wolfram Kahl: GT-VMT 2009.

Languages

- English (native)
- German (beginner)
- French (beginner)
- Chinese (beginner)