A Programming Model and Language for Concurrent and Distributed Object-Oriented Systems

Jan Schäfer

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte

Dissertation

Datum der wissenschaftlichen Aussprache: 5. November 2010

Dekan: Prof. Dr. Karsten Berns
Vorsitzender der Promotionskommission: Prof. Dr. Reinhard Gotzhein
Erster Berichterstatter: Prof. Dr. Arnd Poetzsch-Heffter
Zweiter Berichterstatter: Prof. Dr. Einar Broch Johnsen

D 386
To my family
Abstract

The wide availability of multi-core processors and the ubiquitous presence of the Internet lead to new challenges in software design and implementation. Software has to be written in a parallelizable way to profit from multiple cores. Interaction with distributed Internet services requires coping with message delays and network failures. These challenges reach application domains, like desktop applications, which have been mainly written in a sequential way in the past. The concurrency model of mainstream object-oriented programming languages is based preemptively scheduled threads, which concurrently work on a shared object-heap. This programming model is highly prone to race conditions, i.e., hard to find concurrency-related errors that are not easily reproducible. To synchronize threads and prevent data races, operating system mechanisms like locks have to be used. Experience shows that this programming model is too difficult for most programmers, is not very modular, and is not well suited for the behavioral description of software components. Furthermore, the thread-based model is not appropriate for realizing distributed systems, due to its inherent synchronous communication model.

This thesis proposes a novel programming model for concurrent and distributed, object-oriented systems. The so-called cobox model generalizes the concept of active objects to concurrent, object-oriented runtime components, called coboxes. CoBoxes have their own, local object-heap, which cannot be directly accessed by other coboxes. CoBoxes communicate by asynchronous method calls with standard objects as targets, where multiple objects of a single cobox can be used for interaction. Computations inside coboxes happen via standard object-oriented programming, combined with cooperative multi-tasking. The thesis at hand presents a formalization of the semantics of the cobox model in a core calculus. The calculus is proved type-sound and several additional properties are formally covered. The dynamic semantics of the core calculus is implemented in the rewriting logic framework Maude. In addition, the cobox model is realized in a practical programming language called JCoBox, which extends standard sequential Java. JCoBox is implemented by a Java compiler extension as well as a JVM bytecode rewriter. The performance of JCoBox can compete with state-of-the-art actor implementations for the JVM. The practicability of the proposed programming model and language is evaluated by the design and implementation of several concurrent applications.

Geschwindigkeit der JCoBox-Implementierung ist vergleichbar mit anderen modernen Aktor-Implementierungen für die JVM. Die Praktikabilität des vorgeschlagenen Programmiermodells und der Sprachumsetzung ist evaluiert durch das Design und die Implementierung von mehreren nebenläufigen Anwendungen.
Acknowledgments

First of all I thank my supervisor Prof. Dr. Arnd Poetzsch-Heffter. Besides the financial aspect of organizing the funding of my research, he had given me much freedom in the choice of my research and gave me constant support and motivation. I also thank Prof. Dr. Einar Broch Johnsen who reviewed this thesis as the second reviewer and Prof. Dr. Reinhard Gotzhein for chairing the doctoral committee.

I thank all my past and current colleagues of the Software Technology Group for many joyful days and fruitful discussions. It has always been a pleasure to work in this group. Concerning this thesis, I thank, in particular, Yannick Welsch for greatly supporting me during the last months of the dissertation writing by relieving me of much of the daily project work. In addition, I thank Yannick for a thorough proof reading of this thesis. For further proof reading I thank Ilham Kurnia and Jean-Marie Gaillourdet.

I thank all students, who participated in the cobox journey. In particular, I thank Peter Wortmann for his Scala realization of the cobox model. His work sharpened my view on the interplay between promises, futures, and transfer objects. In addition, he revealed several bugs in the JCoBox implementation and several ideas from his task scheduler implementation greatly improved the performance of the JCoBox scheduler. I also thank Alexander Worret, who was the first student who used JCoBox for a larger case study. I thank Arda Subasi, who initiated the bytecode rewriting tool for JCoBox and also helped with the performance evaluation.

I thank the authors of all the open source software that I used for doing my research, which are far too many to name here. I also thank Handerson Gomes who made the beautiful cover picture available under a free license.

I am very thankful to my parents-in-law Larissa and Eduard for taking care of our daughter in the last month before the finishing of this thesis. Without their support, it would have been much more difficult for me to finish the thesis in time.

I thank my parents Marion and Eugen for everything they did for me.

I thank my daughter Eline Jolie. With her, no day is like the other, and she always reminds me that there are much more important things in life than computer science.

Last but not least, I thank my wife Ekaterina, who always believes in me.
# Contents

1 Introduction
   1.1 Multi-Threaded Object-Oriented Programming Revisited . . . . . . . . . . 2
      1.1.1 Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
      1.1.2 Sequential Programming Model . . . . . . . . . . . . . . . . . . . . . 3
      1.1.3 Thread-Based Concurrency . . . . . . . . . . . . . . . . . . . . . . . . 4
   1.2 Active Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
   1.3 Object-Oriented Components . . . . . . . . . . . . . . . . . . . . . . . . . . 7
   1.4 Goals, Contributions, and Outline . . . . . . . . . . . . . . . . . . . . . . 9
      1.4.1 Goals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
      1.4.2 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
      1.4.3 Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2 The CoBox Model
   2.1 Motivating Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
      2.1.1 Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
      2.1.2 Object-Oriented Design . . . . . . . . . . . . . . . . . . . . . . . . . 14
      2.1.3 Thread-Based Implementation . . . . . . . . . . . . . . . . . . . . . 15
   2.2 Basic Concepts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
      2.2.1 CoBoxes and Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
      2.2.2 Tasks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
      2.2.3 Asynchronous Communication . . . . . . . . . . . . . . . . . . . . . 19
      2.2.4 Designing the Chat Example . . . . . . . . . . . . . . . . . . . . . . 20
   2.3 Synchronization and Data Exchange . . . . . . . . . . . . . . . . . . . . . 21
      2.3.1 Futures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
      2.3.2 Synchronous Communication . . . . . . . . . . . . . . . . . . . . . . 24
      2.3.3 Promises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
      2.3.4 Data Transfer and Object Sharing . . . . . . . . . . . . . . . . . . . 25
      2.3.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
   2.4 Properties . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
      2.4.1 Data Races . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
      2.4.2 Deadlocks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
Contents

2.4.3 Determinism .............................................. 29
2.5 Related Work and Discussion ................................ 31
  2.5.1 Shared-State Concurrency ................................. 31
  2.5.2 Message-Based Concurrency ............................... 35
  2.5.3 Other Programming Models ............................... 41
  2.5.4 Futures and Promises .................................... 42
  2.5.5 Summary and Discussion ................................. 43

3 Core Calculus .................................................. 47
  3.1 Syntax ...................................................... 47
    3.1.1 Abstract Syntax ........................................ 49
    3.1.2 Auxiliary Functions .................................... 50
  3.2 Static Semantics ............................................ 52
    3.2.1 Subtyping ............................................... 52
    3.2.2 Context Conditions ..................................... 52
    3.2.3 Typing Rules ............................................ 53
  3.3 Dynamic Semantics .......................................... 56
    3.3.1 Evaluation Contexts .................................... 56
    3.3.2 Semantic Entities ....................................... 57
    3.3.3 Auxiliary Functions .................................... 58
    3.3.4 Transition Rules ....................................... 59
  3.4 Type Soundness ............................................. 63
    3.4.1 Typing of Semantic Entities ............................. 64
    3.4.2 Auxiliary Functions and Predicates ...................... 65
    3.4.3 Properties of Data Transfer ............................. 67
    3.4.4 Well-Formed Configurations ............................. 68
    3.4.5 Preservation Lemma ..................................... 69
    3.4.6 Progress Lemma ......................................... 70
    3.4.7 Type Soundness ......................................... 73
  3.5 Properties .................................................. 73
    3.5.1 Data Races ............................................... 73
    3.5.2 Determinism ............................................. 74
    3.5.3 Sequential Java Equivalence ............................. 77
  3.6 Discussion and Related Work ............................... 78

4 The JCoBox Language ............................................. 81
  4.1 Core Constructs ............................................. 81
    4.1.1 Object and CoBox Creation ............................... 82
    4.1.2 Asynchronous Method Calls ............................... 83
    4.1.3 Futures ................................................ 83
    4.1.4 Synchronous Communication .............................. 83
    4.1.5 Promises ............................................... 84
    4.1.6 Cooperative Multi-Tasking .............................. 84
4.1.7 Transfer Classes .................................................. 86
4.1.8 Immutable Classes ............................................... 86
4.1.9 Start-Up and Termination of JCoBox Programs ............. 87

4.2 Extensions ............................................................ 87
4.2.1 Timeouts .......................................................... 87
4.2.2 Futures .............................................................. 88
4.2.3 Additional Java Features ....................................... 90
4.2.4 Legacy Java Interoperability ................................. 94
4.2.5 Encoding JCoBox Syntax in Standard Java Syntax ....... 98
4.2.6 Distributed Programming .................................... 101

4.3 Related Work and Discussion .................................... 103
4.3.1 Actor and Active Object Realizations ....................... 103
4.3.2 Discussion, Limitations, and Future Work ................. 105

5 Implementation .......................................................... 107
5.1 Requirements ......................................................... 107
5.1.1 Performance Requirements ................................. 108

5.2 Possible Implementation Options ................................. 108

5.3 Realizing the CoBox Model in a Thread-Based Model ......... 109
5.3.1 The Object–CoBox Relation .................................. 110
5.3.2 How and When to Use Threads ............................... 110

5.4 The JCoBox Runtime .................................................. 112
5.4.1 JCoBox Runtime Access ..................................... 112
5.4.2 Standard Objects ............................................... 113
5.4.3 CoBoxes ........................................................... 113
5.4.4 Asynchronous Calls .......................................... 114
5.4.5 Futures and Promises ......................................... 114
5.4.6 Threads ............................................................ 115
5.4.7 The CoBox Scheduler ......................................... 116
5.4.8 The Thread Pool .................................................. 117

5.5 Java Translation ....................................................... 117
5.5.1 Asynchronous Method Calls ................................. 117
5.5.2 Synchronous Method Calls ................................... 118
5.5.3 Object Creation .................................................. 120
5.5.4 Transfer Classes ............................................... 122
5.5.5 Immutable Classes .............................................. 123

5.6 Extensions ............................................................... 124
5.6.1 Deadlock Detection ............................................ 124
5.6.2 RMI Support ...................................................... 125

5.7 Translation Tools ...................................................... 127
5.7.1 The JCoBox Compiler ......................................... 127
5.7.2 The JCoBox Bytecode Rewriter ............................. 128
## Contents

5.8 Performance .................................................. 129
  5.8.1 Benchmark Programs ................................. 129
  5.8.2 Setup ................................................... 130
  5.8.3 Results ................................................ 130
  5.8.4 Discussion ............................................ 131

5.9 Discussion and Related Work ............................ 132
  5.9.1 Task Cooperation .................................. 133
  5.9.2 Task Execution .................................... 133
  5.9.3 Distributed Programming .......................... 133

6 Practical Evaluation ........................................ 135
  6.1 Describing Systems by CoBox Diagrams .............. 135
    6.1.1 Basic Diagrams .................................... 136
    6.1.2 Multiplicities ...................................... 136
    6.1.3 Node Groups ....................................... 137
  6.2 Writing Concurrent Desktop Applications ............ 137
    6.2.1 CoMusic: Requirements ......................... 138
    6.2.2 CoBox Design ..................................... 138
    6.2.3 JCoBox Implementation .......................... 139
    6.2.4 Discussion ........................................ 145
  6.3 The FourWins Game ...................................... 146
    6.3.1 Requirements ...................................... 147
    6.3.2 Design and Implementation .................... 148
    6.3.3 Discussion ........................................ 152
  6.4 The CoCoME Example .................................... 153
    6.4.1 Informal Description ............................. 153
    6.4.2 Design and Implementation .................... 153
    6.4.3 Discussion ........................................ 154
  6.5 Experience and Discussion ............................ 154

7 Conclusion .................................................... 157
  7.1 Contributions .......................................... 157
  7.2 Discussion .............................................. 158
  7.3 Outlook .................................................. 159

A Core Calculus – Details and Proof ..................... 163
  A.1 Dynamic Semantics .................................... 163
    A.1.1 Substitution ....................................... 163
  A.2 Type Soundness ......................................... 164
    A.2.1 Properties of Data Transfer .................... 164
    A.2.2 Proof of the Well-Formedness Lemma ........... 165
    A.2.3 Auxiliary Lemmas .................................. 167
    A.2.4 Proof of the Preservation Lemma ............... 170

xiv
List of Figures

1.1 Sequence diagrams showing the concept of synchronous method invocation. ........................................ 4
1.2 Sequence diagram showing the effect of monitors in a multi-threaded setting ........................................... 6
1.3 A runtime view to a linked list box ........................................... 8
1.4 Overview over the chapters of this thesis and possible read paths. ........................................... 11
2.1 Conceptional scenario of the chat example. ........................................... 14
2.2 Heap structuring by coboxes ........................................... 17
2.3 Cooperative multi-tasking inside a cobox. ........................................... 19
2.4 Sequence diagram illustrating asynchronous communication in the cobox model. ........................................... 20
2.5 Runtime view of the chat application in the cobox model ........................................... 21
2.6 Sequence diagram illustrating cooperative future claiming. ........................................... 22
2.7 Sequence diagram illustrating exclusive future claiming. ........................................... 23
2.8 The copying of transfer objects, when passed to another cobox. ........................................... 26
2.9 Schematic view of the cobox model. ........................................... 27
2.10 Direct circular dependency between tasks. ........................................... 28
2.11 Indirect circular dependency between tasks. ........................................... 29
5.1 The execution strategy of tasks ........................................... 112
5.2 Class diagram of the JCoBoxCore class. ........................................... 113
5.3 Class diagram of the CoObjectClass class. ........................................... 113
5.4 Class diagram of the CoBoxImpl class. ........................................... 113
5.5 The AsyncCall interface and its implementing classes. ........................................... 114
5.6 Class diagram, showing the implementation of futures and promises. ........................................... 115
5.7 Class diagram of the CoBoxThreadImpl class. ........................................... 115
5.8 Class diagram of the QueueScheduler3 class. ........................................... 116
5.9 Class diagram of the ExecutedBasedThreadPool class. ........................................... 117
5.10 Example code and sequence diagram showing a reentrant synchronous call that leads to a reentrancy exception. ........................................... 125

xvii
List of Figures

5.11 Compilation process when using the JCoBox compiler................. 128
5.12 Compilation process when using the JCoBox bytecode rewriter........ 129
5.13 Performance comparison of JCoBox with other actor implementations 131
5.14 The different benchmark executions on the Xeon-Platform............. 132

6.1 CoBox diagram with two coboxes and three objects.................... 136
6.2 Mapping of a cobox diagram to a cobox diagram with multiplicities..... 136
6.3 Screen shot of the CoMusic application.................................. 138
6.4 Runtime view of the CoMusic application.................................. 139
6.5 Sequence diagram of the CoMusic example................................. 146
6.6 Snapshot of the FourWins application..................................... 147
6.7 Runtime architecture of FourWins......................................... 149
6.8 Design of the CoCoME example............................................ 154

D.1 The pingpong benchmark executed on different platforms............. 192
D.2 The chameneos benchmark executed on different platforms............ 192
D.3 The ringmsgs benchmark executed on different platforms.............. 193
D.4 The ringnodes benchmark executed on different platforms............. 193
List of Tables

2.1 Comparison of different programming models .......................... 45
3.1 Abstract syntax of JCoBox\textsuperscript{C} ................................................. 48
3.2 Functions and predicates on the abstract syntax ....................... 51
3.3 Subtyping rules of JCoBox\textsuperscript{C} ................................................. 52
3.4 Context conditions for JCoBox\textsuperscript{C} programs. ....................... 53
3.5 Type judgments of the static semantics ................................. 53
3.6 Type environments ................................................................. 54
3.7 Expression typing of JCoBox\textsuperscript{C} ....................................... 55
3.8 Evaluation contexts ............................................................... 57
3.9 Semantic entities of JCoBox\textsuperscript{C} ....................................... 57
3.10 Auxiliary functions ............................................................ 59
3.11 JCoBox\textsuperscript{C} cobox-local rules ........................................ 60
3.12 JCoBox\textsuperscript{C} global rules .................................................. 62
3.13 Type judgments for semantic entities ................................. 64
3.14 Typing of semantic entities .................................................. 65
3.15 Well-Formed Configurations ................................................. 69
3.16 Operational semantics for the Java-core language JavaSeq ......... 77
## List of Listings

2.1 The interfaces of the chat example. ........................................... 15

4.1 Implementation of a dating service using promises. ......................... 85
4.2 Treatment of static fields in JCoBox. ........................................... 92
4.3 A simple “Hello World” example, showing how to work with Swing. .... 96
4.4 Non-distributed time server implementation ................................. 101
4.5 Distributed time server implementation ...................................... 102

5.1 Translation of an asynchronous method call in JCoBox to the corres-
ponding Java code. ................................................................. 119
5.2 Example for the implementation of a wrapper method ....................... 121
5.3 Generation of static creation methods for a cobox class. ................... 123
5.4 The RemoteCoObject interface ................................................ 126

6.1 The main method of CoMusic ...................................................... 140
6.2 The Song class........................................................................... 141
6.3 The SongStatus class. ............................................................... 141
6.4 The SongModel cobox class ....................................................... 142
6.5 The SongGUI class and the SongTable class ................................. 143
6.6 The DownProcess class ............................................................. 144
6.7 The BasicPlayer class ............................................................... 144
6.8 The PlayTask class. ................................................................. 145
6.9 Simplified implementation of the Controller class. ......................... 150
6.10 Simplified implementation of the ComputerPlayer class. ................. 151
6.11 Simplified implementation of the HumanPlayer class. ..................... 152

B.1 The sorts of the semantic entities of JCoBox\textsuperscript{M} .................. 178
B.2 The SEMANTICS-CONSTR module of JCoBox\textsuperscript{M}. ................. 180
B.3 The evaluation contexts of JCoBox\textsuperscript{M}. ............................. 181
B.4 The meta-variables used by the rules of JCoBox\textsuperscript{M}. ............... 182
B.5 The cobox-local rules of JCoBox\textsuperscript{M}. ............................... 183
B.6 The global rules of JCoBox\textsuperscript{M} .......................................... 185
List of Listings

C.1 The JCoBox class API. ............................................. 188
C.2 The Fut interface. ................................................ 189
C.3 The future handler interfaces and adapter classes. .......... 190
C.4 The Promise interface. ........................................... 190
List of Theorems, Lemmas, and Definitions

Theorems

3.1 Type Soundness ................................................................. 73

Lemmas

3.3 Well-Formedness Invariant .................................................. 68
3.4 Preservation ................................................................. 70
3.6 Progress ................................................................. 72
3.7 Initial CoBox Typing ...................................................... 73
3.8 There are no Simultaneous Field Accesses ......................... 74
3.9 CoBox-Local Determinism Modulo Futures ......................... 76
3.10 Single-CoBox Determinism ............................................... 76
3.11 Sequential Program Determinism ...................................... 76
3.12 Sequential Java Equivalence ........................................... 78
A.2 Inversion ................................................................. 167
A.3 Substitution ............................................................... 167
A.4 Variable Typing Extension ............................................... 167
A.7 Typing Context ........................................................... 168
A.8 Copy Typing ............................................................... 169

Definitions

3.2 refs ............................................................... 66
3.3 rdom ............................................................... 66
3.4 farrefs .............................................................. 66
3.5 obj ............................................................... 67
3.6 Well-Formed Configurations ........................................... 68
3.7 Well-Typed Configurations ........................................... 69
3.8 Reference Typing Extension ........................................ 69
3.9 Null Access ............................................................ 70
3.10 Far Direct Access ..................................................... 70
3.11 Broken Resolve ......................................................... 70
3.12 Broken Expression .................................................... 71
3.13 Broken Task ............................................................ 71
3.14 Blocked Task ........................................................... 71
3.15 Inactive CoBox ......................................................... 71
3.16 Existing Futures ......................................................... 72
3.17 Resolvable Futures .................................................... 72
3.18 Terminal Configuration ............................................... 72
3.19 Field Access ........................................................... 74
3.20 Simultaneous Field Access ......................................... 74
3.21 Renaming Function $\alpha$ ............................................ 75
3.22 $\alpha$-Renaming ........................................................ 75
3.23 $\alpha$-Equivalence ..................................................... 75
3.24 Sequential Program ................................................. 76
3.25 Java-like Program ...................................................... 77
3.26 Configuration Equivalence .......................................... 78
CHAPTER 1

Introduction

During the last 45 years, software development has been driven by Moore’s Law [Moo65]. Software developers could rely on the fact that the performance of sequential software doubled approximately every two years. Whereas the chip industry is still able to double the transistor density at this rate, they cannot significantly increase the clock rates of their chips anymore. Instead, the number of cores per chip is increased [Cre05]. Even standard desktop computers today are multi-processors with four or more CPU cores. Increasing performance in this new setting is not for free anymore [Sut05]. Applications have to be explicitly written in a parallelizable way to benefit from future processor generations [SL05]. Software developers must now deal with Amdahl’s Law [Amd67], which roughly means that the performance of a program is mainly restricted by its sequential part. But multi-core processors are not the only challenge that programmers are confronted with. The omnipresent Internet is pushing software into the distributed setting. An increasing number of applications are either written in a distributed way, or interact with web services, which require programmers to deal with network latency and failure.

These challenges reach application areas, like desktop applications, which have been mostly written in a sequential, single-threaded way in the past [Cre05]. Such applications are typically written in object-oriented programming languages (OOLs) like C++ [Str00], Java [GJSB05], or C# [ECM06]. The concurrency model of these languages is mainly based on the underlying operating system (OS) concepts of preemptively scheduled threads that concurrently work on a shared heap. This low-level programming model is unsafe by default as it allows threads to access the same memory locations simultaneously. The programmer has to use mechanisms like locks to prevent data races. Locks, however, do not guarantee data race freedom, introduce a great risk of non-deterministic deadlocks, and are not very modular [Jon07, Lee06]. Experience shows that such a programming model can mainly be handled correctly by a few experts [Ous96, Gus05]. Besides its complexity in the local case, the programming model of standard OOLs is also not well-suited for distributed programming. Typically, distributed programming is based on remote
method invocation (RMI), which tries to transparently map the thread-based model to the distributed setting. This synchronous communication model, however, leads to strong performance problems in the presence of network latencies [AG03]. In addition, the thread-model is not completely transparent in the distributed setting, due to different reentrancy semantics of threads, for example [HMRT03]. Thus it is difficult to port existing thread-based applications to the distributed setting and requires a substantial rewrite as well as a completely different conceptional thinking.

As concurrency and distribution are now getting into mainstream programming, new concurrency models are needed, which abstract from the thread model and which raise concurrency to a higher level. Such models must be safe by design, i.e., data races must not be possible. This is of extreme importance as data races appear nondeterministically, are very hard to find, are not easily reproducible, and, in particular, cannot be addressed by standard testing [Lee06]. In addition, a concurrency model must allow for writing modular software. This means that it must be possible, in such a model, to reason about the behavior of components in isolation, without taking the concurrency of the environment into account. Finally, the concurrency model must be seamlessly applicable to the distributed setting, without a conceptional breach. The goal of this thesis is to develop, formalize, implement, and evaluate a concurrency model for object-oriented programming, which satisfies these criteria.

Overview. The remainder of this chapter is structured as follows. In Section 1.1, the thread-based programming model of standard OOLs is reviewed and discussed. Section 1.2 presents an alternative concurrency model for object-oriented programming (OOP), namely the active object model. Section 1.3 introduces a notion for object-oriented runtime components, which raises the unit of state and behavior from single objects to groups of objects. Finally, Section 1.4 presents the goals, contributions, and outline of this thesis.

1.1 Multi-Threaded Object-Oriented Programming Revisited

This section discusses and reviews the multi-threaded concurrency model of OOP and its realization in modern OOLs. This discussion focuses on strongly-typed, class-based OOLs.

1.1.1 Objects

The standard computation model of OOP is based on objects exchanging messages [Arm06]. Objects have a unique identifier, a state, consisting of the values of its attributes (also called fields), and a behavior, which defines the way an object reacts to certain messages. The identifier of an object can be used as an address to send messages to the object. An identifier that is used to refer to an object is called a
1.1 Multi-Threaded Object-Oriented Programming Revisited

References can be stored in the local state of objects and can be passed to other objects by messages. In addition, every object knows its own identifier and can thus pass references of itself to other objects. In class-based languages, each object has an associated class, from which it was instantiated. The class defines the type of the object, the types of its fields, and a set of methods. The methods define the possible messages that an object can receive. In addition, they provide the implementation of the behavior of an object, when receiving the corresponding message. Objects are typically reactive, that is, they are passive until they receive a message. As a reaction to a message, an object can create new objects, can update its state, and can send messages to other objects. In addition, an object can return a response to the object from which the message originates.

1.1.2 Sequential Programming Model

Whereas the literature in general speaks of objects that exchange messages, e.g., in Smalltalk [GR83], the actual realization in typical OOLs is different. Rather then exchanging messages, objects invoke methods on other objects. Method invocation, however, is in general synchronous, which means that the sender has to wait until the receiver has finished the method execution. This leads to a strong temporal coupling between the sender and the receiver as the sender cannot proceed until the method call has returned [Jon07]. Technically, a method invocation results in a new stack frame that is put onto the current execution stack. The sending stack frame, becomes dormant until the method returns.

Example. This principle is shown in the left sequence diagram (a) of Figure 1.1. A thread that is currently executing code of object \( a \) invokes a method call on object \( b \). The current stack frame becomes dormant, and a new stack frame is created for executing the method. When the method is finished, the previous stack frame becomes active again.

Reentrancy

In addition to the strong coupling, standard OOLs permit reentrant calls. A reentrant call happens when a thread with dormant stack frame in an object \( a \), executes a method on an object \( b \) and then invokes a method on \( a \) again. The right sequence diagram (b) of Figure 1.1 shows this by example. This means that between the initial invocation of a method and its return, another method can be invoked on the caller object, potentially observing or even mutating its state. Reentrancy greatly complicates the behavior of objects [MSL99, MTS05]. In a sequential setting, reentrancy has to be allowed, because otherwise it would be impossible to write programs with mutually dependent objects.
1.1.3 Thread-Based Concurrency

As already stated, concurrency in most OOLs is based on threads. Threads are preemptively scheduled, independently running, control flows. All threads share a global state, which can be equally read and modified by all threads. Threads run independently from each other until they explicitly synchronize by acquiring a lock, for example. The problem of the thread-based concurrency model is that introducing a single additional thread to an otherwise sequentially written program, completely invalidates all existing assumptions of the sequential program until otherwise justified. It is impossible to safely introduce concurrency into a sequential program if the sequential program is not implemented in a thread-safe way. The reason for this dilemma is simple: in a purely sequential program, the state is solely modified by a single thread. This thread owns this state and it has the guarantee that no thread can modify this state. Or, in the words of Tom Hickey:\footnote{http://clojure.org/state}

\textit{“the world is stopped while you look at or change it”}.

Introducing multi-threading completely destroys this illusion because the state is not owned by a single thread anymore. This is why multi-threaded programming is much harder than sequential programming [Ous96]. In fact, multi-threaded programming is fundamentally different to sequential programming. Although a single thread runs in a sequential, deterministic manner, the behavior of multiple, concurrently running threads is highly non-deterministic.

The thread-based model has another, more conceptional, problem: threads are completely unrelated to objects. As threads may “jump” from object to object, it is difficult to describe and understand the behavior of the object itself. In particular, an object cannot be treated as a unit of behavior. This fact becomes even more complicated in the presence of reentrancy.
Monitors and Locks

To prevent certain thread interleavings, locks and monitors [Han73, Hoa74] are in general used in practice.

Monitors. In the object-oriented setting a monitor protects a single object from being accessed by several threads concurrently. The first thread that enters a monitor gains access to the monitor and then has exclusive access to the protected object. Other threads that try to enter the monitor have to wait until the monitor is free again. Synchronous communication in OOLs now becomes a large problem [LHG86]. The reason is that if a thread that holds a monitor $m$ invokes a method on an object protected by a monitor $m'$, the thread blocks $m$ for other threads while it waits for the result of the method call. This quickly leads to deadlocks in recursive scenarios [CH88] and also leads to the nested monitor problem [Lis77, Had77]. To prevent deadlocks, monitors are often reentrant. Reentrancy allows a thread that owns a certain monitor to call into other monitors and if the same thread then issues a callback to the first monitor, it is not prevented from doing so.

The sequence diagram of Figure 1.2 shows such a scenario. It consists of three objects, which are all protected by a different monitor. Two threads, $t_1$ and $t_2$, are running in parallel, and are executing code of objects $a$ and $c$, respectively. Thread $t_1$ now invokes a method on object $b$. As $t_1$ still owns the monitor of $a$, $a$ is blocked for other threads. The monitor of $b$ is currently free, thus $t_1$ can enter it and can invoke the method. In the meantime, thread $t_2$ also tries to invoke a method on $b$. As $b$ is currently owned by $t_1$, it has to wait. Now $t_1$ issues a callback on $a$. This callback is permitted, although the monitor is currently blocked, because the monitor is reentrant and allows thread $t_1$ to enter it as it owns this monitor. After $t_1$ has returned to $b$ and then returned to $a$, the monitor of $b$ becomes free and $t_2$ can enter it. The example shows that, like in the sequential setting, reentrancy makes it possible to observe and potentially modify the state of objects during a method call.

Locks. Instead of using monitors, which protect objects, it is also possible to use locks (or semaphores [Dij02]) instead. Locks allow the programmer to protect a certain code region (a critical section), from multiple thread interleavings. This concept is thus not directly related to objects. Whereas it allows for a more flexible synchronization than monitors, this mechanism is unsafe by default, i.e., data races can still happen. Another problem of critical sections is their code-centrality. As a critical section only protects a region of code and not a memory region, it is possible that two threads, which both are in different critical sections, can still access the same memory location concurrently, leading to data races. Thus, in order to avoid data races it requires non-local reasoning [VTD06].

Locks and Monitors in Java and C#. OOLs like Java and C# mix the monitor concept with locks. Each object has its own lock, but it is not enforced to acquire this lock
when accessing the state of the object. In addition, the usage of the object lock is not restricted to the object itself, but it is possible by other objects to acquire the lock externally, which is not possible with standard monitors. Java and C# provide *synchronized blocks* (called *lock statements* in C#), which acquire a lock and ensure that the lock is always released when leaving the block, even in the case of exceptions. It is also possible to declare methods as *synchronized*, which is syntactic sugar for enclosing the method body by a synchronized block.

### 1.2 Active Objects

A programming model which better matches the idea of objects that exchange messages is the *actor model* [Agh86], first developed in the context of artificial intelligence [HBS73]. Actors are similar to objects as they have unique identifiers and exchange messages. But in contrast to objects, actors are inherently concurrent. The behavior of an actor is typically defined by a *script* (or *body*). At runtime, an actor consists of a single thread that executes the script and a *mailbox* that acts as an (unbounded) buffer for received messages. In response to a message, an actor can create new actors, can send messages to other actors, and can change its behavior, which, in particular, means that an actor can define which messages to accept. The first realizations of actors have been done in a functional setting [Lie87].

Actors have no methods. They typically accept messages by using pattern matching. When actors have methods, and are thus more similar to objects, we call them *active objects* [Car93]. Active object approaches typically use *asynchronous method calls* as a communication mechanism. Unlike actor messages, these calls are *type-safe*, i.e., it

---

Note that in the literature there is no strict distinction between the terms actors and active objects.
is statically clear that these messages are understood by the active object at runtime.

The active object model does not suffer from the problems that are apparent in the multi-threaded model. The key properties of the active object model are:

- sequential computation inside an active object has standard sequential invariants as the state of an active object can only be modified by a single thread, in fact the execution thread has atomic access to the state of the active object;

- communication is asynchronous and thus active objects are temporally decoupled, leading to loosely-coupled systems, with a lower risk for deadlocks;

- the behavior of active objects can be described and understood locally without taking the environment into account;

- messages do not carry thread-identifiers and there is no reentrancy complexity; and

- distributed computation is naturally modeled as there is no shared state.

The pure active object model also has some disadvantages. As every object is concurrent, concurrency is often too fine-grained. The state of active objects is limited to the values of its fields, which is often not enough as objects, in general, rely on other objects for holding additional state [NVP98]. Finally, the single-threading of active objects makes it difficult to realize multiple independent control flows, which is important for combining reactive with active behavior [JOY06], for example.

1.3 Object-Oriented Components

The unit of state and behavior in object-oriented programs is the object. An object encapsulates data and defines the possible operations permitted on that data. Typically, however, an object relies on other objects to hold additional state and implement additional behavior [NVP98, PS98]. A simple example is a linked list object that uses internal node objects to implement the link structure and where iterator objects can be used to access the list (see Figure 1.3). Thus, state and behavior is often realized by interacting object groups and not by single objects. These object groups actually form conceptional components at runtime. These components, however, are implicit and only exist in the mind of the programmer.

To make these components explicit we introduce the concept of boxes [PHS06, PHS07]. A box is, like objects, a runtime entity with a unique identifier and represents an object-oriented runtime component. The state of a box is constituted by an arbitrary number of objects. During the lifetime of a box, the number of objects is not fixed, i.e., new objects can be added, and other objects can become garbage. Objects, however, belong to a single box for their entire lifetime. Thus it is always clear to which box the behavior of an object belongs. A box defines a clear boundary at runtime, which divides all objects into objects belonging to the box (internal objects) and objects of the box environment (external objects). In addition, a box can publish
objects to its environment by exposing their reference. Such objects can be considered as services that the box provides. We thus call such objects service objects in the following. The set of service objects of a box forms its runtime interface. All objects that are not published are called local objects. Every box has an initial service object, which is the first object of a box. In the list example, the LinkedList object is the initial object, the Iterator objects are service objects, and the Node objects are local objects. All other objects belong to the environment and are thus external objects.

Boxes have no influence on the semantics of a program. Their main intent is to describe runtime component boundaries. This component boundary, however, can be used for different purposes. For example, it can be used to describe the behavior of boxes by tracing the messages that cross the boundary at runtime [PHS07]. Boxes can also be used to enhance information hiding for dynamic components by restricting casts at runtime [PHGS08], or to improve object encapsulation [PHGS07]. The important point of boxes is that they lift the unit of state and behavior from single objects to runtime components.

The concept of boxes can be used in the sequential as well as in the multi-threaded setting. However, the behavior of boxes in a multi-threaded setting can become very complex and, in particular, requires to include thread-identifiers in messages, similar to Ábrahám et al. [ÁGS08]. Having to deal with thread-identifiers in messages has several drawbacks. First, in the standard setting of reentrant locks and monitors, the acceptance of a message might depend on the invoking thread. Thus, a component has to distinguish reentrant calls from other calls. This leads to scenarios, where the behavior of a callback depends on the concrete thread which does the callback. And second, thread identifiers in messages reveal the internal concurrency of a component as the environment can distinguish two calls if they come from different internal threads.
One of the goals of this thesis is to take the idea of boxes and generalize it to the concurrent setting. But instead of directly transferring the box concept from the sequential setting to the multi-threaded setting, this thesis uses boxes as the basis for a new concurrency model for OOP. This concurrency model essentially lift the concept of active objects to concurrent boxes, i.e., to coboxes.

1.4 Goals, Contributions, and Outline

The goal of this thesis is to develop a new concurrency model for object-oriented languages. This model should have certain properties, which are briefly given in Section 1.4.1. The major contributions of this thesis are summarized in Section 1.4.2. Finally, Section 1.4.3 outlines the contents of the thesis.

1.4.1 Goals

The concurrency model that is developed by this thesis should have the following properties, ordered by importance.

- It should be safe by design, i.e., data races must not be possible in the model.
- It should fit into the object-oriented paradigm. This means that the model should be based on objects.
- It should be component-based, i.e., it should be possible to describe the behavior of object-oriented components, independently of thread identifiers.
- It should be modular, i.e., it should be possible to develop and test object-oriented components and be able to compose them without invalidating their invariants.
- It should be compatible with distributed computation.
- It should allow for writing ordinary sequential OO components and combining them with concurrent ones in a safe way.
- It should be practical, i.e., it must be efficiently implementable and should allow for interaction with existing legacy code.
- It should require little effort to understand and use the model for programmers that are used to standard OOLs.

1.4.2 Contributions

This thesis develops a concurrency model for object-oriented languages based on a notion of object-oriented runtime components. The main contributions of this thesis are:
• The development of a novel concurrency model based on the idea of boxes and active objects.
• The formalization of the model in a type-sound core calculus and the formal investigation of core properties of the calculus.
• The implementation of the core calculus in the rewriting logic framework Maude.
• The realization of the model as a practical Java language extension called JCoBox.
• The implementation of a compiler and runtime for JCoBox with a performance that is competitive with state-of-the-art actor implementations.
• The development of an embedding of JCoBox in a Java-compatible syntax and the implementation of a bytecode rewriter for that embedding.
• A practical evaluation of the cobox model and JCoBox by showing how to design and implement concurrent object-oriented programs with JCoBox.

Parts of these contributions are already published in previous works by the author [SPH10a, SPH10b, SPH08].

1.4.3 Outline

This thesis develops a model and language for concurrent, distributed, object-oriented systems. Chapter 2 explains the programming model in detail. Chapter 3 provides a core calculus, precisely describing the static and dynamic semantics of the model and formally addressing several core properties. Chapter 4 shows how the programming model is realized in a practical Java language extension. The implementation of the language is presented in Chapter 5. How the programming model and language can be used in practice is presented in Chapter 6. Finally, Chapter 7 concludes.

Figure 1.4 gives an overview over the chapters of this thesis and possible ways how to read it. For example, it is possible to skip Chapter 3 if one is not interested in the formalization of the cobox model. Chapter 5 can also be skipped if one is not interested in the details of the implementation of JCoBox.
Figure 1.4: Overview over the chapters of this thesis and possible read paths. Dashed lines denote that the connected chapters serve as supplemental information, which can be used for reference.
This chapter describes the cobox model, the programming model that is proposed by this thesis. The chapter starts by presenting an example in Section 2.1 to motivate the central design decisions of the cobox model. The basic concepts of the cobox model are explained in Section 2.2. These include the runtime structure, tasks, and asynchronous communication. Section 2.3 describes how tasks can be synchronized in a data-driven way by using futures and promises and how coboxes can exchange data. Section 2.4 presents the core properties of the cobox model. Finally, Section 2.5 discusses the cobox model in the context of related work.

2.1 Motivating Example

To illustrate and motivate central design decisions of the cobox model, we use a simple IRC-like chat application as an example.

2.1.1 Requirements

The chat application consists of multiple chat clients communicating with a single chat server. A user interacts via a GUI with the chat client. The server can have multiple chat rooms. Clients can connect to a server by using a unique login name. After a client has successfully connected, it can join or open chat rooms. Clients that have joined a chat room can publish messages to that room. Published messages are broadcast to all clients which joined the same room. A client can be in several chat rooms at the same time. Clients and servers typically run on different distributed machines connected via a network. In this example, we abstract from the distribution aspects of the application, but assume that there is a transmission delay between clients and server and that clients can silently disconnect from the server.

Figure 2.1 shows an exemplary runtime configuration of the chat application. Two clients are connected to the server. The server has currently three open rooms, where
Chapter 2 The CoBox Model

The first room has one joined client and the second room has two joined clients.

Implementation Challenges

Implementing such an application has some challenges. On the client side, the GUI and the network-related parts should be decoupled. Otherwise network latencies may affect GUI responsiveness. User input and network events happen non-deterministically and must be coordinated. The client must handle different chat rooms and has to relate them to the server-side chat rooms. In addition the client must manage the server connection.

The server has to deal with concurrently accessing clients, each with possible different connection statuses. Multiple chat rooms must be managed and the server must know which clients have joined which chat room. The order of messages appearing at a chat room should be consistent in all clients. The communication with different clients should not interfere with each other, in particular, they should not block each other. Finally, the server should scale with the number of clients and chat rooms.

2.1.2 Object-Oriented Design

The next step is now to find an object-oriented design for our example application. At this point we ignore the concurrency aspects and only try to model the relevant interfaces as if we would model a sequential application. For brevity, we only look at the parts of the application which participate in the interaction between client and server. For the client and the server we define interfaces shown in Listing 2.1. The Client interface only has a single method to set the server. The Server interface has a method to connect to the server, which takes a login name as parameter and returns a ServerSession interface. The ServerSession interface is a typical service interface, which is used by the client to interact with the server after it has connected. By using a separate session object for each client, the server can easily associate method calls to clients and track their session state. Without using such service objects, most methods of the server interface would require an additional parameter to identify the
2.1 Motivating Example

Sending client and had to use internal maps to store the client associated state, which complicates the server logic.

```
interface Server {
    ServerSession connect(String name);
}
interface Client {
    void setServer(Server server);
}
interface ClientSession {
    void newMsg(ChatMsg msg);
    void userJoined(String user);
    void userLeft(String user);
}
interface ServerSession {
    RoomSession join(String room,
        ClientSession session);
    String[] getAllRooms();
    void alive();
    void disconnect();
}
interface RoomSession {
    void publish(String msg);
    void leave(String reason);
}
```

Listing 2.1: The interfaces of the chat example.

The `ServerSession` interface has methods which can be used by the client to disconnect from the server, to get the names of all available rooms, and to join a certain room. In addition, the interface has an `alive()` method, which must be constantly called by the client to signal that it is still connected. The `join` method takes a room name and a `ClientSession` as parameter and returns a `RoomSession`. For each chat room, the client can use a different `ClientSession` object, so that the client can easily differ the messages of different rooms. The `ClientSession` interface is used by the server to send messages to the client that have been published in the joined chat room. Such messages are represented by `ChatMsg` objects, which carry the message content and the name of the user that sent the messages. The `RoomSession` object that is returned by the `join` method is used by the client to send messages to the joined room. Messages are simply sent as `String` objects as the room session knows the originating client. This also prevents that a client can send messages under a wrong name. Finally, the client can also leave the room with the `leave` method.

2.1.3 Thread-Based Implementation

Having defined the basic interfaces of our application, we can now start with its implementation. We assume an object-oriented language with a standard concurrency model based on preemptive threads.

The main question is which threads are required to realize the application. The client is mainly triggered by events from the GUI, which runs in its own thread.
But the client must ensure not to block the graphical user interface. So it at least requires an additional thread that decouples the network interaction from the GUI. This requires that events that are coming from the GUI and are executed by the GUI thread must be given to the network thread. Standard method calls cannot be used as they are executed by the calling thread. The client also receives concurrent calls from the server on different client session objects, which all have to be coordinated. Finally, at least one additional thread is required by the client to constantly signal aliveness to the server. All in all the implementation of the client is far from being straightforward and is much more complex than what would one expect at first sight.

The chat server has no GUI and thus concurrent accesses only come from the clients. However, multiple chat rooms with several service objects must be managed. As monitors only protect single objects, care must be taken not to introduce data races. The server must communicate with many clients and has to ensure that communications are correctly ordered and do not influence each other.

### 2.2 Basic Concepts

The basic idea of the model that is proposed in this thesis stems from the active object model: distributed, stateful entities run concurrently and communicate via asynchronous messages. This basic idea is combined with the idea of boxes (cf. Section 1.3): instead of only having single objects as concurrent entities, the cobox model has object-oriented components, called coboxes, as concurrent entities. A cobox is similar to a box as its state consists of multiple objects, where some of them are local objects and others are service objects and can be used to interact with the cobox. Like in the active object model, the state of a cobox cannot be accessed by other coboxes. CoBoxes can only communicate by asynchronous method calls on each others service objects. As opposed to active objects, coboxes are not single-threaded. Instead, coboxes can have multiple threads, called tasks. To avoid data races, tasks are scheduled cooperatively. This section describes the basic concepts of the cobox model, i.e.,

- how the state of coboxes is represented in terms of objects,
- how the behavior of coboxes is expressed in terms of tasks, and
- how coboxes communicate in terms of asynchronous method calls.

#### 2.2.1 CoBoxes and Objects

The central idea of the cobox model is that there is no global object-heap, but that the heap is distributed over coboxes. Each cobox has its own local object-heap, which cannot be accessed by other coboxes. Like objects, coboxes are runtime entities with a unique identity. However, coboxes cannot be referenced directly, only the objects of a cobox can be referenced. At creation time it is decided to which cobox an object
2.2 Basic Concepts

(a) flat object graph  
(b) cobox-structured object graph

Figure 2.2: A flat object graph (a) versus an object graph structured by coboxes (b). Far references, i.e., references to objects of different coboxes, are indicated by a dashed, instead of a solid, line, which represent near references.

belongs to. Objects are owned by their cobox for their entire lifetime, i.e., objects cannot move to other coboxes. CoBoxes are dynamically created together with an initial object. The initial object can then be used to interact with the cobox. This means that every cobox has at least one object.

From the perspective of a cobox, an object can either be internal if it is owned by the cobox, or external if it is owned by a different cobox. Following the naming of the E programming language [MTS05], we call internal objects near objects and external objects far objects. Analogous to object roles, we call a reference near if it refers to a near object and far if it refers to a far object. Whether a reference is near or far is important in the cobox model. A near reference can be used like references known from standard OOLs, i.e., it can be used to access fields or directly call methods. A far reference, however, can only be used as a target for asynchronous method calls.

Besides the distinction into near and far objects, objects are either service objects or local objects. Service objects are objects that have been published by the cobox and are thus accessible by other coboxes; local objects are only accessible inside the same cobox. Note that this means that far references can never refer to local objects.

Example. Figure 2.2 illustrates the cobox model by comparing the object graph with a standard global heap model. Figure 2.2(a) shows the typical global heap model, Figure 2.2(b) shows the object graph in the cobox model, where the objects are structured by coboxes. Near references are indicated with a solid line and far references with a dashed line.

2.2.2 Tasks

Objects are passive entities, they exist and wait until they receive a message. If a cobox only consists of objects, for example, in its initial state, it is also passive, i.e., it only waits for messages. A cobox becomes active if one of its objects receives a
message. In that case, a new task is created in the cobox that owns the receiver object. The task is then responsible for executing the body of the corresponding method. Tasks are similar to standard threads as they have a local state and represent control flows. Unlike threads, tasks are not orthogonal to components. Instead, tasks are owned by a single cobox for their entire lifetime, i.e., they never leave their cobox. A task is executed in a standard sequential imperative manner. It can directly access all objects of its cobox, i.e., it can access their fields and can directly invoke methods. Such direct method calls are immediately executed by the calling task in the standard stack-based way.

**Cooperative Multi-Tasking**

Tasks of the same cobox are scheduled cooperatively (see Figure 2.3). Cooperatively means that there can only be one active task, and the active task has to explicitly give up control to allow other tasks to become active. Tasks cannot be interrupted preemptively. The active task has thus exclusive access to the cobox-local state, i.e., to the state of all objects of the cobox, while it is running. If the active task never gives up control until it terminates, it is executed in a standard sequential way. If a task is not active, it is idle. An idle task is either ready or suspended. Initially a task is in the ready state. Ready tasks are organized in a FIFO queue (the ready queue). When the active task gives up control, the next task from the ready queue becomes active. A task can give up control in three ways: it can terminate, it can yield, which immediately adds it to the end of the ready queue, and it can suspend. A suspended task always waits for a future (see Section 2.3.1). When the future is resolved, the task is resumed and added to the ready queue, for eventual execution.

**Tasks vs. Threads**

The task concept differs from typical thread-based approaches, because tasks are bound to components and threads are not. Tasks in the cobox model cannot leave their cobox. The internal task structure of coboxes is thus not visible to the environment. This simplifies the externally visible behavioral of a cobox and hides internal implementation details. In addition, the cobox model does not suffer from the reentrancy problem of standard threads (cf. Section 1.1.2). As a task cannot leave its cobox, it cannot reenter it.

Tasks in the cobox model run completely isolated to each other until they explicitly cooperate. This makes it possible to understand the behavior of a task in isolation, without regarding the behavior of other tasks or take the concurrency of the environment into account. Nevertheless, interleaving of tasks is still possible at programmer defined scheduling points. This is important to be able to combining active and reactive behavior and to support the flexible delegation of method calls and non-blocking waiting for futures [dBCJ07], without being forced to use an event-based programming model.
2.2 Basic Concepts

2.2.3 Asynchronous Communication

CoBoxes communicate by asynchronous method calls. As coboxes cannot be referenced, the targets of asynchronous method calls are the service objects of a cobox. As explained in Section 2.2.1, objects of other coboxes are referenced by far references. Far references can only be used to asynchronously invoke methods, i.e., to send messages\(^1\). It is not possible to use far references for direct method calls or fields accesses.

Asynchronous method calls are similar to standard method calls in OOP. They have an object as target and can have several argument values. Asynchronous method calls do not block the calling task. The calling task proceeds without waiting for the result of the call. This leads to a temporal decoupling of caller and callee, where the execution of the asynchronous method call on the caller side is completely independent of the state of the callee. In contrast to synchronous calls in the multi-threaded setting, the calling task does not leave the cobox. Instead, a new task is created in the target cobox. That task is then responsible for executing the method call.

Example. Figure 2.4 illustrates the concept of asynchronous method calls and tasks by means of a sequence diagram. It shows two coboxes, each owning a single object. There is one task, \(t_1\), running in the left cobox which asynchronously invokes two methods, \(m_1()\) and \(m_2()\), on object \(b\) of the right cobox. The call \(m_1()\) results in a new task \(t_2\), which immediately becomes active. The second call \(m_2()\) arrives at \(b\), while \(t_2\) is still active. The call results in a new task \(t_3\), which, however, cannot immediately become active, as \(t_2\) is still running. Thus it stays in the ready state. Note that, while this happens, task \(t_1\) concurrently keeps running in the left cobox. Eventually \(t_2\) terminates and \(t_3\) becomes active. It then issues a call \(n()\) back to object \(a\). As \(t_1\) is still active, the new task \(t_4\) starts in the ready state and becomes active when \(t_1\) terminates.

Properties. Asynchronous method calls have the following properties: (1) they are always delivered eventually and (2) they are ordered. The ordering of messages is

\(^1\)In the following, the term message is used as a synonym for an asynchronous method call.
only guaranteed between two coboxes. It is thus a partial ordering, which guarantees that two subsequent calls from one cobox, targeting objects of another cobox, are executed in the given order. This also holds if the calls are on different objects, as long as these objects are owned by the same cobox.

### 2.2.4 Designing the Chat Example

Designing the chat application in the cobox model is more or less straightforward. The application is already naturally divided into concurrently running entities, namely the different clients and the server. In a first step each of these entities would be realized as a separate cobox. Communication between the clients and the server then happens asynchronously, which already ensures that they do not block each other. The client cobox should further be split into at least two coboxes: one for handling the user input and one for interacting with the network. This ensures that the GUI is decoupled from the rest of the client application. The server can also be split into several coboxes, namely one server cobox and one cobox for each chat room. Thus, different chat rooms are decoupled from each other and can run concurrently, which increases the scalability of the server.

The server needs to track client sessions. In the cobox model this can be realized by creating a service object for each client. After the client has connected, it gets this service object from the server. The client then communicates with the service object instead of the server object. Client-local session state can then be managed by this object. The chat room coboxes are designed in a similar way. Whenever a client joins a chat room it gets a chat session object from the chat room, which can be used by the client to send messages to the chat room. The client itself also has to manage chat sessions with different chat rooms, which is ideally handled by separate objects.
2.3 Synchronization and Data Exchange

Figure 2.5 shows a runtime view of an example configuration. It consists of two clients. The first client has joined a single chat room, the second client has also joined that chat room and is also in an additional chat room. Not shown are local objects and objects for realizing the data that is transferred between the different coboxes. From the runtime view, the potential concurrency is immediately clear. For example, the Client and ClientSession objects share a common cobox. This allows tasks of this cobox to exclusively access the state of all its objects. Due to the asynchronous communication between the different coboxes, the components are loosely coupled. As the GUI of the client lives in its own cobox, communication with the network related parts does not block the GUI. No additional mechanisms need to be used to achieve this. The previously chosen object-oriented design (cf. Section 2.1.2) can be implemented directly in the cobox model, which shows that the cobox model fits very well with object-oriented programming.

2.3 Synchronization and Data Exchange

CoBoxes can pass object references to other coboxes as arguments of asynchronous method calls. But so far there is no mechanism for getting the result of asynchronous method calls, synchronizing tasks, and exchanging data between coboxes. This section explains how these issues are addressed in the cobox model.
2.3.1 Futures

To be able to respond to asynchronous method calls, asynchronous method calls return futures \cite{BH77, Hal85, YBS86, Lie87, NSS06, ÁGGS09}. Futures are placeholders for values that are eventually computed. A future can be in two states: unresolved or resolved. Initially, a future is unresolved. Once a future is resolved to a value, it stays resolved forever. To retrieve the value of a future, it has to be explicitly claimed \cite{ÁGGS09}. Claiming a future can be done in two ways: exclusively and cooperatively.

Cooperative Claiming

If the active task claims a future cooperatively, the task gives up control of the cobox and is added to the suspend set until the future is resolved. When the future is resolved, the suspended task is resumed and added to the ready queue to become active eventually.

When cooperatively waiting for the future, the programmer has to guarantee that the cobox is in a consistent state as other tasks may observe the cobox state while the task waits. In addition, when a waiting task is activated again, the state of the cobox might have been changed by other tasks. Speaking in terms of verification conditions, the invariants of the cobox must hold before a cooperative wait, and after the wait, only the invariants of the cobox can be assumed.

Example. Figure 2.6 illustrates the cooperative future claiming with a sequence diagram. It shows two coboxes, each having a single object. The left cobox has two tasks, \( t_1\) and \( t_2\), where \( t_1\) is active and \( t_2\) is ready. Task \( t_1\) sends a message to object \( b\) of the right cobox. This results in a new future \( f\) in the left cobox. Task \( t_1\) then
awaits the future, i.e., claims the future cooperatively. Task $t_1$ immediately becomes suspended, allowing task $t_2$ to become active. Task $t_3$ then terminates and resolves future $f$ with value $v$. Task $t_1$ is thus resumed and becomes ready. Finally, task $t_2$ terminates and task $t_1$ becomes active again.

**Exclusive Claiming**

If the active tasks exclusively claims a future, the active task stays active, i.e., does not give up the control of the cobox, until the future is resolved. This means that in the meantime no other task of the cobox can be activated, i.e., the whole cobox is blocked. This has the advantage for the active task that the state of the cobox cannot be changed while it is waiting for the future value. In addition, it also means that the current state of the cobox cannot be seen by any other task. This allows the waiting task to leave the cobox in an inconsistent state, i.e., a state where not all invariants are satisfied, without fearing that this state can be observed by other tasks.

**Example.** Figure 2.7 shows a similar scenario to Figure 2.6, but now task $t_1$ claims the future exclusively. This means that, while future $f$ is not resolved, task $t_1$ blocks the cobox for other activities. Thus task $t_2$ cannot be activated and has to stay in the ready state. When $f$ is resolved, task $t_1$ continues its execution until it eventually terminates and allows task $t_2$ to become active.

**Discussion**

Whether a future should be claimed exclusively or cooperatively depends on the given scenario. As a general rule, an exclusive claim should only be used if the invoking
task cannot establish the invariant of its cobox, without knowing the value of the claimed future. In addition, exclusively waiting should only be done if the future, which the task waits for, can be resolved by the called method, without a callback to the waiting cobox. As in this case the callback can never be executed as the waiting cobox is blocked, a deadlock would occur. When using the cooperative variant, the claiming task must establish the invariant of the cobox before the claim, because other tasks could be activated in between. Futures are the only way in the cobox model to synchronize tasks, which leads to a data-driven synchronization [CHS04]. Futures are first-class values and can be passed to other coboxes as arguments or results of asynchronous method calls.

2.3.2 Synchronous Communication
Asynchronous method calls and futures can be used to simulate synchronous communication by immediately claiming the future of an asynchronous method call. The caller then has to wait until the callee has finished. But contrary to standard synchronous communication, the caller can decide whether to wait cooperatively or exclusively. If it waits cooperatively, other tasks of the cobox can be activated during the synchronous call, and, in particular, reentrant calls are possible. If the caller waits exclusively, no other tasks can be activated in the meantime, and, in particular, reentrant calls are impossible.

2.3.3 Promises
Futures cannot only be resolved implicitly by asynchronous method calls, but also explicitly using promises [MTS05, NSS06, ÁGGS09]². A promise is, like a future, a place holder for a value that is eventually computed. A promise, however, can cannot be read directly, it can only be resolved. To read the value of a promise one has to get a future that is associated with the promise. That future can then be used to obtain the value. A promise can thus be seen as a value sink, whereas a future is a value source [Wor09b].

A promise can have multiple associated futures. When the promise is resolved, it resolves all associated futures to the corresponding value. As promises can only be resolved once and reading promises can only be done by futures, promises can be safely shared between coboxes. Promises are a flexible communication and synchronization mechanism and can be used to model many synchronization patterns, like condition variables, for example.

Promises are also used internally by the cobox model to realize asynchronous method calls with futures. When a method is invoked asynchronously, an implicit promise is created, which is resolved by the task that executes the call. In addition, a future from that promise is retrieved, which is used as the result of the asynchronous

²The terms future and promise are not used consistently in the literature. This paper uses definitions similar to Ábrahám et al. [ÁGGS09] and Niehren et al. [NSS06].
call. When the task that executes the called method terminates, it does not directly resolve the future, but it resolves the corresponding promise, which in turn resolves the future. In addition, if a future is passed to another cobox, the future is not passed by reference, but a new future is created in the target cobox, which is associated with the promise of the original future. Thus futures are never directly shared between coboxes.

2.3.4 Data Transfer and Object Sharing

Standard objects in the cobox model are passed by references to other coboxes. These objects act then as service objects for the originating cobox. Only passing references between coboxes, however, is not sufficient in many cases. In general, it should be possible to transmit complex data. For example, the ChatMsg object of the chat example (cf. Listing 2.1) is actually data and should not act as a service object. For this purpose the cobox model adopts two approaches. Beside standard objects, the cobox model has two additional kinds of objects, namely transfer objects and immutable objects. Transfer objects are copied between coboxes, immutable objects are shared between coboxes.

Transfer Objects

Transfer objects are special objects, which are passed by copy to other coboxes. The target cobox then gets a near reference to the new object copy instead of a far reference to the original object. The copy is transitive, i.e., the object itself is copied and all transfer objects, which are referenced by that object are transitively copied. Transfer objects can never be passed by reference to other coboxes. Thus transfer objects can never be service objects for a cobox and are always local to their cobox. Transfer objects are similar to serializable objects in Java RMI [Ora10a].

Example. Figure 2.8 shows an example of the transitive copying of transfer objects. In the example, transfer object $d$ is copied, due to an asynchronous method call of task $t_1$ on object $g$, denoted by $g!m(d)$. The left side shows the system state before the copying, the right side after the copying. As $d$ references transfer objects $b$ and $c$, they are also copied to the target cobox. Standard objects that are referenced by transfer objects, like $a$ and $f$ in the example, are copied by reference. As the example shows, near references can become far references and vice versa.

Immutable Objects

As transferring data using transfer objects can be inefficient due to the copying overhead, it is possible to directly share state between coboxes using immutable objects. Immutable objects [HPSS07, CWOJ08, Blo08] never change their state after their construction, and it is safe to access their state concurrently. Like standard objects,
immutable objects are passed by reference between coboxes. Conceptually, immutable objects are not owned by any cobox. Nevertheless, references to immutable objects are treated as near references. It is thus possible to directly call methods and to access the fields of immutable objects. Immutable objects can only reference standard objects or other immutable objects. Transfer objects, as well as futures to transfer objects, cannot be referenced by immutable objects.

Note that immutability is only shallow, i.e., objects referenced by immutable objects can be mutable again. In the context of the cobox model this kind of immutability is enough as it guarantees thread-safety.

2.3.5 Summary

Figure 2.9 gives a complete picture of the cobox model. It shows that the state of a cobox consists of a set of objects, which can be standard or transfer objects, and a set of futures. Transfer objects and futures can only be referenced by near objects, standard objects can be referenced by far and near objects. Immutable objects do not belong to the state of a cobox, but are treated as near objects by coboxes. Immutable objects can reference standard objects, but cannot reference transfer objects. Promises do not belong to the state of a cobox. They can be referenced by objects and are resolved by tasks. Futures are always associated to a promise, which is responsible for resolving them. Futures of the same promise can belong to different coboxes. The behavior of a cobox is defined by its tasks. A cobox can have multiple tasks, of which at most one is active and the others are either suspended, i.e., waiting for a future, or are ready, i.e., waiting to become active.
2.4 Properties

2.4.1 Data Races

The cobox model is inherently free of data races. This is due to the fact that coboxes do not share any mutable state. CoBoxes are only allowed to communicate via messages. The state of a cobox can only be directly accessed by its own tasks. As no two tasks of one cobox can be active at the same time, concurrent access to state is not possible. A task has even exclusive access to the whole state of the cobox until it gives up control.

However, high-level data races [AHB03] can still happen in two different scenarios. The first scenario is that a task yields between two state accesses. Between both accesses, other tasks may become active and could change the state. It is thus important that before a task gives up control, it must ensure that the state of the cobox is consistent. After it is activated again, it can only make the assumption that the state is consistent, but it cannot assume anything else.

The second scenario for high-level data races comes from the fact that if messages are sent to a cobox by more than one cobox, the messages can arrive with different possible interleavings. Consider a cobox with a single counter object which has methods to increase its value and to obtain its value. If the counter cobox is shared by several other coboxes and one of these coboxes sends two subsequent messages to obtain the counter value, the second result value can be arbitrarily higher than the first value because other coboxes could have increased the counter in between.

2.4.2 Deadlocks

The risk of deadlocks in the cobox model is in general much lower than in the standard thread-based model. This is due to the fact that coboxes communicate
asynchronously. However, as tasks of coboxes can wait for the result of asynchronous method calls by using futures, deadlocks can still happen in the cobox model.

**Direct Task Dependencies**

Whenever a task waits for a future, it might be the case that this future can never be resolved due to a circular dependency. As futures are first-class values, it is possible that tasks mutual wait for each others futures. Figure 2.10 shows such a scenario. Task \( t_1 \), which is responsible for resolving future \( f_1 \), waits for future \( f_2 \) which is resolved by task \( t_2 \), which in turn waits for future \( f_1 \). In the scenario, both tasks wait cooperatively for each future, so both are suspended. This shows that in such a scenario there can still be tasks running in the cobox. Only the tasks that are involved in the dependency are dormant forever. If a task waits exclusively for a future, the whole cobox is blocked forever. Note that such direct dependencies between tasks are very unlikely, because it requires to store a future in a field, which is then used by another task to wait for.

![Figure 2.10: Direct circular dependency between tasks. Two tasks mutual wait for futures, which they resolve.](image)

**Indirect Task Dependencies**

If a task exclusively waits for a future, deadlocks can even happen if tasks do not directly depend on each other. Figure 2.11 shows such a scenario. Task \( t_1 \) exclusively waits for future \( f_2 \), thus preventing any other task to become active in its cobox, while it is waiting. Future \( f_2 \) is resolved by task \( t_2 \), which in turn waits for future \( f_1 \), which is resolved by task \( t_3 \). Task \( t_3 \) lives in the same cobox as task \( t_1 \). Hence, although task \( t_3 \) is ready for execution, it cannot be activated as task \( t_1 \) blocks the cobox. We thus have a vicious circle, i.e., a deadlock.
Such indirect dependencies are much more likely than direct dependencies. They happen for example in synchronous call-back scenarios, where a task synchronously calls a method in another cobox, which then synchronously calls back to the original cobox. Note, however, that indirect task dependencies can only happen when exclusively waiting for a future.

2.4.3 Determinism

Concurrent programming models are in general non-deterministic in order to allow for a parallel execution. The more non-determinism is allowed by a model, the more parallelism is in general possible. Non-deterministic execution, however, does not mean that the results are undetermined. For example, purely functional programming languages like Haskell [Has10] allow for a highly non-deterministic evaluation of expressions, without sacrificing the determinism of its result value. As long as the evaluation is confluent [CHS04], the result is still deterministic. In imperative languages, however, non-deterministic program execution, often also lead to non-deterministic results. For a limited set of configurations, active object systems can be confluent [CHS04], for example, if the communication structure at runtime is organized as a directed tree. These confluence results can be lifted to the cobox model, so that it is possible to define criteria under which a cobox system is deterministic. However, this thesis has not further investigated this issue. The following subsections precisely describe where the cobox model introduces non-determinism and where it is deterministic.
Non-Determinism in the CoBox Model

In general, the programmer is only interested that the visible behavior of a program is deterministic. If a program is not completely deterministic, it is important that the programmer is aware of all possible non-determinism that can occur. The cobox model is non-deterministic in the following aspects:

1. Tasks of different coboxes are executed concurrently.
2. Messages are only ordered with respect to two coboxes.
3. Futures are resolved non-deterministically.
4. Suspended tasks can be resumed out of order.

The first aspect means that steps of active tasks of different coboxes can be arbitrary interleaved or even executed in parallel. This, however, is harmless because tasks of different coboxes run isolated from each other. The interleavings with tasks of other coboxes are completely irrelevant to a task as it does not share any state with them. The behavior of a task can only be affected by other tasks when they explicitly wait for a future or yield, which is an explicit operation that the programmer is aware of. The second aspect introduces non-determinism because messages are only partially ordered. If there are two coboxes, which both send a message to a third cobox, it is undetermined which message is handled first. However, all messages sent by a cobox to the same cobox are executed in the given order. This ordering is the best ordering that can be efficiently implemented in a distributed setting. A total ordering of all messages is not possible to implement in a scalable way. The third source of non-determinism is that futures are resolved non-deterministically. For example, if two futures are associated to the same promise, it is not determined which of these futures is resolved first. In fact, both futures can be resolved in parallel.

Determinism in the CoBox Model

Execution in the cobox model is deterministic in several aspects, which are the following:

1. Tasks are executed deterministically until they explicitly give up control.
2. Cooperative scheduling is deterministic.
3. Messages are ordered with respect to two coboxes.

Most importantly, the active task of a cobox has atomic access to the state of the whole cobox until it explicitly releases control. During such execution frames, tasks are executed in a standard, sequential, deterministic way. This makes it possible
to locally reason about the behavior of tasks. In addition, programmers used to
standard OOP can directly reuse their knowledge and can rely on the fact that
sequential code behaves in the cobox model like in standard OOLs. The second
point is that the cooperative scheduling of tasks is deterministic and done in a FIFO
way. For example, a task that yields only becomes active again after all other tasks
of the ready queue have become active in between. This also ensures fairness of
the local scheduling of tasks. Finally, as already mentioned above, asynchronous
messages are partially ordered. This guarantees that two subsequent messages that
have the same source and target coboxes are executed in the order they have been
sent. Our practical experience shows that this guarantee is very important in the
object-oriented setting. As often messages trigger state changes, the ordering of such
messages must be preserved. Without such a guarantee, the sending task is enforced
to explicitly wait for the result of the previous call to ensure that the next call is
executed after the previous one. This, however, greatly reduces possible concurrency
and resembles synchronous communication. In addition, in the presence of message
delays, waiting for each message is an expensive operation. Furthermore, when
ordering is guaranteed, messages can be pipelined [LS88], which is not possible if
one waits for the results of previous messages. The alternative way of guaranteeing
ordering by using message counters [Agh86] should not have to be implemented by
the programmer manually each time he or she requires ordering.

2.5 Related Work and Discussion

There exists a long history of approaches to solve the concurrency problem in
object-oriented languages (surveys and general design choices can be found in
[Ame89, BGL98, Phi00, Has00]). In general, most approaches can be divided into
two categories: (1) approaches that are based on threads that work on shared
memory, and (2) approaches that are based on isolated processes that communicate
by message passing. Some approaches combine both models. Finally, there are
some approaches that cannot be put in either category. There exist many further
concurrency models, for example, based on process calculi, which are not related
to object-oriented programming. This section focuses on approaches that handle
concurrency in OOP.

2.5.1 Shared-State Concurrency

The concurrency model typically used in many programming languages is based on
shared state. The concurrency model of threads, which is used in most mainstream
OOLs, is described in Section 1.1.3. Multiple threads are sharing a common state and
communicate by reading and modifying this state. To achieve mutual exclusion of
state accesses, synchronization mechanism must be provided. This subsection focuses
on approaches that are based on the shared-state multi-threading model. Already
Cooperative Multi-Tasking

To avoid the problems of preemptive threads, but still support multiple control flows, cooperative multi-tasking (also called cooperative multi-threading) can be used [Gus05]. Like in the thread model, tasks work on shared state. Unlike threads, cooperative tasks can neither be preemptively interrupted nor can run in parallel. They have to explicitly give up control to allow other tasks to become active. In addition, at most one task can be active at the same time, all other tasks are suspended. Cooperative multi-tasking is often integrated into languages by coroutines [Con63, MI09]. Coroutines are, for example, used in Simula 67 [DMN70, BDMN75] and Modula-2 [Wir77]. But recently regain attention and are used in novel languages like io [Io 10] and Lua [IdC06], for example.

The advantages of cooperative multi-tasking is that data races are not possible in this model and that no synchronization mechanisms are needed to ensure this. In fact, it is an example of a concurrency model that is safe by design. The number of possible task interleavings is drastically reduced compared to preemptive multi-threading, which makes it easier to reason about programs written in that model.

The obvious disadvantage of cooperative multi-tasking is that multiple processors cannot be utilized as tasks cannot be executed in parallel. Another disadvantage is that tasks can completely block other tasks if they are not cooperative. Fairness has to be guaranteed by the programmer.

Event-Based Programming. A special case of cooperative multitasking is event-based programming [Ous96]. Event-based programming is essentially cooperative task management with manual stack management [AHT02]. In that model a task must terminate in order to allow other tasks to be executed. Cooperation is thus only done by termination. The model consists of a single event-queue, which holds pending events. A single thread executes the events of the event-queue one after the other. During the execution of an event new events can be added to the end of the event-queue for later execution. The execution of event-based systems is very efficient as no locking and context switches are involved.

Beside the general problems of cooperative multi-tasking, event-based programming has additional disadvantages. As the control flow of a program is scattered over event-handlers, programs are difficult to read and understand [vBCB03]. In addition, there are problems with exceptions and the model suffers from the so-called lost continuation problem [FMM07].

Typical applications which are written in an event-based style are programs with graphical user interfaces and servers that use an asynchronous IO mechanism.
Combining Cooperative with Preemptive Threads

*FairThreads.* Boussinot [Bou01, Bou06] proposes FairThreads, which allow for linking a set of threads to a common scheduler. Threads linked to the same scheduler are scheduled in a cooperative way and can communicate by generating and waiting for events, which may have values. Threads are not directly suspended when generating events, but have to execute a special cooperate statement. The crucial point about FairThreads is the scheduling algorithm. Scheduling is done in so-called *instants*. An instant can be considered as a time scope for events. During an instant it is guaranteed that all threads “see” all events generated during that instant and are possibly resumed. If no thread can be resumed by events generated during the current instant, the next instant is started and all remaining events of the previous instant are lost.

Boussinot and Dabrowski [BD06] combine cooperative with preemptive multi-threading similar to FairThreads. Threads can link to schedulers and are then scheduled cooperatively. However, communication between cooperative threads is solely by using a common memory. A type and effect system statically divides the memory into public and private areas, where public areas can be accessed by different threads, only when linked to a scheduler, and private areas can be accessed only by a single thread. The presented language is not object-oriented, and has no concept of asynchronous communication.

Transactions

Recently, the idea of transactions, know from databases [HR83], are integrated into programming languages. Whereas database transactions support the ACID properties [HR83], transactions in programming languages only support the A and I namely, atomicity and isolation. Atomicity means that either all effects of a transaction are applied or none, isolation means that parallel transactions cannot influence each other.

*Transactional Memory.* One of the first approaches was the introduction of transactional memory (TM) [HM93]. A TM is a specially protected memory region, which can only be accessed by transactions. The system then guarantees that transactions are executed atomically and in isolation. Transactional memory can be implemented in several different ways. It can be solely implemented by a software library, [ST95, HG06], or can be implemented in hardware [HM93]. A combination of a software and hardware implementation is also possible [HLM06].

*Language Integration.* Typically, transactions are integrated into languages by means of atomic blocks [HF03, HMPJH05]. Code that is executed inside an atomic block is treated as a transaction. Atomic blocks solve several problems of locks. The most important advantage is that the complete state that is accessed within a transaction
is protected by the transaction. This guarantees that code that runs inside an atomic block is always mutual exclusive to code of other atomic blocks. The programmer does not need to mentally assign memory regions to certain locks. Thus, as long as code is included in an atomic block, access to memory is always safe. Another advantage compared to locks is that atomic blocks are modular. It is possible to nest atomic blocks without taking a certain locking order into account. Deadlocks are not possible when using atomic blocks, but transactions may be aborted.

Discussion. Whereas the basic idea of transactions seems appealing at first glance, and it avoids many problems of locks, integrating the concept of transactions into a programming language raises many questions. One question is how to handle irreversible I/O operations and other side-effects inside transactions [Har05] and how to support communication of threads within transactions [SKBY07].

Transactions do not guarantee absence of data races if the language does allow access to shared memory locations outside of transactions. Also the performance of transactional memory implementations is still a challenging problem [CBM+08]. Transactions do not address one of the main problems of thread-based concurrency, namely synchronous communication. This means that atomic blocks suffer from the reentrancy problem of standard threads (cf. Section 1.1.2). Transactions are also orthogonal to component boundaries, which makes it difficult to describe the behavior of components in that context. In summary, transactions are a very useful concept, but they are not a silver bullet. They only solve a certain part of the thread problem. Transactions seem to be mainly suited for data-centric applications, but they are especially not well suited for writing loosely-coupled component-based applications. An interesting future topic is how transactions can be integrated into the cobox model.

Guava

Guava [BST00] is a Java extension, which guarantees data race freedom. Guava is based on the standard Java thread-based concurrency model, but introduces mechanisms to prevent data races. Similar to the cobox model, Guava divides objects into three categories: Monitors, Values, and Objects. Monitors can be shared between threads and are always thread-safe by enforcing that all methods of Monitors are synchronized. Values have a pass-by-copy semantics like transfer objects. Objects are guaranteed to be never shared by two threads. The latter is achieved by an ownership type system. Value classes are regarded as immutable if they have no update method, i.e., no method that may mutate the state of the object. In that case Value objects are passed by reference. Monitors are similar to coboxes in the sense that they can have a complex internal state consisting of multiple objects. However, multiple service objects are not possible, communication is not asynchronous, and cooperative multitasking is not supported.
Emerald

Emerald [BHJL07] is an object-based language designed for distributed programming. Its main feature is the mobility of objects, allowing for moving objects between locations. Emerald has a notion of object groups, where objects can be attached to other objects. When an object is moved to another location, all attached objects are moved, too. The concurrency model of Emerald, however, is based on threads, synchronous method invocation, and monitors. Emerald has (unchecked) immutable objects.

Sequential and Parallel Object Monitors

*Sequential Object Monitors* (SOM) [CMET04] separate the scheduling of requests from the class that handles requests. This separation of concerns allows a flexible combination of different schedulers for the same class. A SOM can have multiple waiting requests, but tasks of a SOM cannot be executed in an interleaved way. This makes it impossible to have SOMs that act as active objects (even though the authors claim that their approach is close to the active object model). In addition, multiple service objects are not possible with SOM. Having different request schedulers is a useful extension to the cobox model.

*Parallel Object Monitors* (POM) [CMPET08] generalize SOM to allow for multiple, concurrently running threads in a single POM, giving up the guaranteed mutual exclusion to allow for more efficient implementations. Besides the scheduling of request, a POM can also control the reentrancy of calls. A POM can also protect multiple objects for realizing monitors with multiple service objects.

2.5.2 Message-Based Concurrency

Most related to the cobox model are message-based concurrency models. These models are typically characterized by having no shared state and communication is typically asynchronous. Traditionally, many approaches with this concurrency model are targeting distributed systems. But lately, it becomes more and more clear that message-based concurrency is also very useful in the local setting.

Active Object Approaches

Section 1.2 already describes the actor [HBS73, Agh86] and the active object model. The basic idea of these models is that concurrency is structured by data. They can thus be seen as *data-centric* concurrency models. The actor model has been realized, for example, in the languages Act/1 [Lie87] and ABCL/1 [YBS86]. The first commercially successful implementation has been done with Erlang [Arm03]. Lately, the actor model regains much attention, and has been implemented as libraries or extensions for several programming languages, e.g. Scala Actors [HO09]. Contrary to actors, which exchange messages, active object approaches use asynchronous
method calls. First approaches that integrated active objects into programming languages have been POOL2 [Ame89], Eiffel/ [Car93], and SALSA [VA01]. All these approaches have in common that the basic concurrency units are single objects.

ABCL

ABCL [Yon90] represents a family of actor languages. ABCL/1 is a lisp-dialect that integrates the notion of actors. ABCL/1 actors have an ordered message buffer, a local state and a script that defines the behavior. The messages that an actor accepts are specified by pattern matching, which goes through the message buffer and tests for a matching message. Messages can be accepted in two different modes: dormant and waiting. In the dormant mode, all messages that do not match are discarded. In the waiting mode, these messages are kept. ABCL/1 supports three different ways to send messages: the past type, the now type, and the future type. The past type is a simple asynchronous message send. The future type is an asynchronous message send that has a future as a result. Futures in ABCL/1 are queues of values. Futures are not first-class values and cannot be passed to other actors. The now type is a synchronous message passing, which immediately waits for a reply after having sent a message. This can be compared to an asynchronous method call in the cobox model with an immediate exclusive wait for the future. ABCL/1 allows the receiver to continue its execution after it has replied to a method call. In addition, a message can be delegated to another actor, which then has the responsibility of replying to the method call. Message transmission is partially ordered like in the cobox model. ABCL/1 also supports so-called express messages. If such a message is received by an actor, the current activity is preemptively interrupted and the express message is handled. As this introduces the possibility of data races, ABCL/1 has a kind of atomic block, which is guaranteed to be executed atomically. ABCL/1 has a notion of Object Groups, which group a number of objects (actors) together. Such an object group can have an arbitrary number of objects, but only a fixed number can be used to interact with the group. The grouping mechanism of ABCL/1 is only used for debugging purposes.

ABCL/M [Yon90] is a modification of ABCL/1, which includes a notion of unserialized objects. Such objects can process messages in parallel, but have to be completely stateless.

POOL2

POOL2 [Ame89] is an object-oriented programming language that realizes the active object model. POOL2’s basic communication is synchronous. The sender has to wait until the receiver accepts the method call by an explicit answer operation. Like in ABCL/1, the receiver can reply to the sender, but still continue its execution. POOL2 also supports asynchronous communication, which is regarded as syntactic sugar for creating an intermediate object that buffers communication, which, however, does
not preserve ordering of messages. POOL2 has no grouping mechanisms and only one task can be active in an active object.

**Hybrid**

Hybrid [Nie87, Nie92] is an object-oriented programming language with a concurrency model based on *domains*. A domain is similar to a cobox as it owns a set of so-called *dependent* objects. Inside a domain only a single *activity* can be active at a time, similar to cobox tasks. Activities are created by invoking special so-called *reflex* operations. Otherwise, communication between domains is done by synchronous method calls. To avoid deadlocks, Hybrid allows reentrant calls, which makes the behavior of domains dependent on thread identifiers. Interleaved activities inside a domain are possible by a delegation mechanism that frees a domain, while waiting for the result of the delegated call.

**SALSA**

SALSA [VA01] is a programming language designed for the development of dynamically reconfigurable open distributed applications. It is based on the standard active object model and has special features for distributed computing such as the movement of actors between computation nodes. Computations in SALSA are built by using different kinds of continuations, which allow for a flexible ordering of computations and message sends. Like the cobox model, SALSA realizes messages by type-safe asynchronous method calls, but without futures.

**Jac**

Jac [HL06] is an approach to specify concurrency mechanisms in Java in a declarative way, but only at the granularity of single objects. A precompiler transforms special Javadoc annotations into Java code. Only objects that are created by a new expression annotated with *@controlled* are protected by Jac, otherwise all annotations are ignored. Their system has strict mutual exclusion as default, but allows methods that are declared to be *compatible* to be executed concurrently. Active objects can be realized by a special *@auto* annotation to specify methods that should be automatically started when the object is created. Thread synchronization is possible by guards. Methods may be declared to be asynchronous in which case the method is executed by a separate thread when called and a future object is returned. Replacing the default FIFO scheduling strategy is possible by writing manual scheduling code. Jac cannot protect object groups with multiple service objects, and does not guarantee the absence of data races.
Chapter 2 The CoBox Model

ASP

ASP [CHS04] is a calculus that extends the $\zeta$-calculus [AC96] by active objects and futures. Concurrency is introduced in ASP by activities. Activities are similar to coboxes in several aspects. First, the state of activities can consist of multiple objects. Objects in ASP are either active or passive, which correspond to service and transfer objects in the cobox model, respectively. Second, communication in ASP happens by asynchronous method calls that are invoked on active objects. These calls return futures, which are first-class values in ASP. However, unlike coboxes, activities can only have a single active object, multiple service objects are not possible. In addition, ASP does not support multiple tasks inside an activity. Only one task can exist. Activities can have a service method, which can be compared to an actor script. It can be used to specify a concrete communication protocol by using the Serve operation, which specifies the methods to be accepted.

Contrary to the cobox model, futures in ASP are not explicitly claimed. Instead, they are implicitly resolved when their value is needed. This also means that a task in ASP always blocks its activity when it hits an unresolved future. The ASP model is deterministic for a certain kind of confluent ASP systems [CHS04]. The ASP calculus is implemented in ProActive [BBC+06].

Communicating Event Loops

The E programming language [MTS05] introduces a concurrency model called communicating event loops. The unit of concurrency in E is a vat, which hosts a group of objects. All objects of a vat can be referenced by other vats, allowing multiple service objects, equally to coboxes. Communication between vats is based on asynchronous method calls on service objects of other vats. Asynchronous method calls return futures (called promises). Promises in E cannot be explicitly claimed. They can either be used as targets for asynchronous method calls again, or handlers can be registered that are executed in the context of the vat when the future is resolved. Computations inside a vat are only executed by a single thread. This leads to an event-based programming model, where the control flow often is spread over several event handlers. E only guarantees ordering of messages with respect to single objects (in fact even only with respect to single references), whereas the cobox model guarantees ordering with respect to coboxes (cf. Page 30), which we believe is what a programmer in general expects.

AmbientTalk/2. AmbientTalk/2 [VC08, VCMB+07] is a language for mobile ad hoc networks. The concurrency model of AmbientTalk/2 is essentially that of E, but in addition has isolates, which correspond to transfer objects in the cobox model.
2.5 Related Work and Discussion

**Creol**

Creol [JBKO09, dBCJ07, JO07, JO04] is an object-oriented language for concurrent, distributed objects. It has an executable semantics, implemented in the rewriting framework Maude [CDE+07]. Communication is based on asynchronous method calls with futures. Similar to the cobox model, an object may have multiple tasks called *processes*. Scheduling of processes within the same object is handled cooperatively using explicit release points. Processes can synchronize on conditions by using guards, which can also be used to wait for futures. Waiting for futures can be done exclusively and cooperatively. The scheduling and synchronization concepts and its combination with futures of the cobox model are in fact a simplified variant of that of Creol. The unit of concurrency in Creol are single objects. Internal state is solely represented by using functional data types. Multiple service objects are not possible in Creol and asynchronous method calls are unordered.

**Concurrent Smalltalk**

Concurrent Smalltalk [YT86, YT87] is an extension of Smalltalk with asynchronous method calls and so-called *CBox* objects, which are equivalent to futures. So-called *atomic objects* ensure that only one thread can be active inside these objects.

**SR and JR**

SR [ACE+88] and its object-oriented Java-based successor JR [KGMO04] are languages for programming distributed systems. Their concurrency model combines method invocation with message passing. Methods can be invoked either synchronously or asynchronously using two different invocation statements. In addition, the receiver can serve method calls by two different ways: either by a *proc* statement or by one or more *in* statements. Where the *proc* statement creates new processes, the *in* statement is used to wait for certain method calls. JR does not guarantee data race freedom.

**Kilim**

Kilim [SM08] provides a library and a JVM bytecode rewriting tool to realize actors in Java by light-weight continuations. Communication between actors is realized by typed *mailboxes*, which is incompatible with standard method calls. A flow sensitive type system allows for transferring tree-like object structures between actors without the need of copying, a mechanism, which could be integrated into the cobox model. Actors are executed by schedulers. Cooperative scheduling of actors can be achieved by configuring the corresponding scheduler to be single-threaded.
Thorn

Thorn [BFN+09] is a scripting language targeting the JVM. It features lightweight, single-threaded, isolated processes, which communicate via message-passing. Similar to SR and JR (cf. Section 2.5.2), Thorn combines method invocation with message sending. However, only methods declared to be asynchronous can be invoked in an asynchronous way. In contrast to the cobox model, Thorn does not support multiple tasks within a process. However, Thorn has a special \texttt{splitSync} construct, which allows method calls to be forwarded without blocking the process. Multiple service objects are not possible in Thorn.

Axum

Axum [Axu10] is a programming language for the .NET platform [Mic10], with an actor-like programming model. In Axum different \textit{agents} run concurrently and communicate solely by messages that are send via channels. Channels can have input and output \textit{ports}. The behavior of channels can be specified by state-machines using \textit{protocols}. Messages are restricted types of objects, where the concrete form can be specified by so-called \textit{schemas}. Messages are either deep-copied between agents or are passed by reference when they are immutable. Agents in Axum can share state if they belong to the same \textit{domain}. A domain is a collection of fields, methods, and agents. The agents of a domain are distinguished into \textit{reader} and \textit{writer} agents, with corresponding access rights to the state of their owning domain. Multiple reader agents of the same domain can run in parallel, but a writer agent runs exclusively in a domain. Axum allows the definition of side-effect free functions, which can be used in pipelines for data-flow like computations.

Concurrent Aggregates

\textit{Concurrent Aggregates} (CA) [CD90] are a mechanism to hierarchically structure active objects. A CA can be used as an object again. As CAs are \textit{unserialized}, i.e., can process multiple messages in parallel, they allow for an efficient distribution of messages. CAs cannot have multiple service objects.

Join Calculus

The Join calculus [FGL+96, FG96] is a calculus for describing distributed agents with explicit locations. The programming model is based on a functional language with so-called \textit{join patterns}. The Join calculus is implemented as extensions for several object-oriented languages, for example Co\textsubscript{o}[BCF02], and Join Java [IJ03, vI05]. These approaches, however, have in common that the underlying concurrency model is based on state-sharing threads. Join patterns are used for synchronization, but data race freedom is not guaranteed. Join patterns are orthogonal to the underlying concurrency model and could as well be integrated into the cobox model.
2.5.3 Other Programming Models

There are some programming models and languages which do neither fall into the standard thread-based category, nor are pure message-based approaches. This subsection presents such models.

**Singularity OS**

In the Singularity OS \[FAH^+06\] processes are strongly isolated and communicate via asynchronous messages. Each process has its own memory heap, where a single so-called *exchange heap* exists for transferring memory between processes. All programs for Singularity OS have to be written in the language Sing#, which is an extension of C#\[ECM06\]. Sing# allows the definition of channel contracts to specify bi-directional protocols. Their system can statically check that programs follow these contracts. A form of object ownership, called *tracked structs*, allows the safe transfer of object structures between processes without the need of copying, using the exchange heap. Processes and coboxes have in common that both have their own heap. In contrast to coboxes, communication in Sing# is done via message channels, instead of using object references. This has the advantage that it is easier to verify protocol usage, but has the disadvantage that it is not as flexible. The main difference between processes in Singularity OS and coboxes is that the former use the standard thread and synchronization mechanisms to support concurrency within a single processes, whereas the latter use cooperating tasks.

**SEDA**

SEDA \[WCB01\] structures an application into event-driven *stages*, which are connected by explicit queues. Each state can be single-threaded or multi-threaded depending on the chosen scheduler. SEDA focuses on server applications to realize high throughput of requests, it is not a programming model for general applications.

**X10**

X10 \[SJ05, CGS^+05\] is a novel object-oriented language targeting non-uniform cluster computing. X10 has special support for explicitly partitioning the heap by a concept called *places*. Inside a place, multiple *activities* can run concurrently. Activities are bound to their owning place. The state of a place can only be accessed by activities that are located in that place. To access state of other places, a new activity has to be spawned in the targeting place. State inside a place can be accessed concurrently by multiple activities. To prevent data races X10 offers several synchronization constructs, like atomic blocks, for example. To allow for data-parallel algorithms, X10 supports *distributed arrays*. Finally, X10 supports futures, parallel loops, and *clocks* for realizing *barriers*, and *value classes* for realizing immutable objects.
The X10 programming model is similar to the cobox model in several aspects. Places can be compared with coboxes in the sense that they partition the object heap into groups of objects, which can only be accessed locally in a place. Activities are similar to tasks because they are bound to their place and cannot “jump” to another place. But there are also some differences. Places in X10 cannot be dynamically created. Instead, their number is fixed for the entire execution. Places also do not communicate by asynchronous method calls, but rather by creating asynchronous activities in other places. Both mechanisms can be mutually encoded, but we believe that asynchronous method calls in general better fit into object-oriented programming. Finally, activities in X10 are not cooperatively scheduled, like in the cobox model, but run preemptively.

2.5.4 Futures and Promises

The terms future and promise are not clearly defined in the literature. The first usage of the term future is due to Baker and Hewitt in the context of Act/1 [BH77], which is a language that realizes the actor model [HBS73, Agh86]. Friedman et al. [FW78] uses the term promise instead of future, but essentially refers to the same concept. Futures in these contexts have the following characteristics: they are implicit, and they are transparent. Implicit means that a computation that needs the value of a future is suspended automatically and implicitly until the future is resolved, in which case the future is replaced by the value, and the computation is resumed again. This mechanism is also called wait-by-necessity [CHS04]. Transparent means that introducing futures into an existing program does not change its semantics, which is always the case in the absence of side-effects [WJH05].

Implicit futures have been used in MultiLisp [Hal85] to realize asynchronous computations. As MultiLisp has side-effects, futures are not transparent. Liskov and Shira [LS88] use the term promise instead of future to denote strongly typed futures. Contrary to implicit futures, their promises have to be claimed explicitly to obtain their value. We thus call these kind of futures explicit futures. Explicit futures are never transparent.

In standard active object approaches that have futures, waiting for a future blocks the active object for other activities. While sometimes useful, this is often undesirable as it reduces concurrency and disallows callbacks. The possibility to wait for futures without blocking, was introduced by Creol [JOY06]. Creol supports guards that suspend the current activity until a condition becomes true, which can, in particular, be the condition that a future is resolved. While an activity is suspended, other activities can run in the active object. Another way of non-exclusively waiting for a future is by using event handlers that are executed when a future is resolved, as done, for example, in the E programming language [MTS05] by so-called when-blocks. Standard synchronous communication cannot be modeled by this mechanism. To model synchronous communication it must be possible to explicitly wait for a future.

Often futures are not explicitly resolved. They are rather bound to an asynchronous
computation and are implicitly resolved when the computation has finished. In the E programming language [MTS05] and Alice ML [NSS06], for example, futures can be explicitly resolved. Futures are explicitly resolved by using a separate object that is associated with the future. Like in Ábrahám et al. [ÁGGS09], that object is called a promise in the cobox model, but it is also called future handle [NSS06] and resolver [MTS05]. Promises are write-once objects, which resolve their associated futures when being written. Promises can not be read directly. Instead, the associated futures have to be used to retrieve their value.

### 2.5.5 Summary and Discussion

There are many concurrency models and languages for object-oriented programming and this section only presents a brief overview. The approaches can be roughly divided into ones that are based on the standard thread-model, and ones that are based on the message-passing model. It is our strong opinion that only the message-passing model is well suited for component-based object-oriented systems. It better matches the idea of components to be the unit of state, behavior, and composition. In the concurrent setting, it is a natural consequence to treat components as a unit of concurrency. This can be considered as a structured approach to handle concurrency, opposed to the unstructured concurrency of using threads. Knowing the component structure directly allows one to understand the concurrency of the system, a fact which is hidden when using a thread-based approach.

The cobox model is based on the message-passing idea. It uses object-oriented components as the unit of concurrency. Components only communicate via asynchronous messages, targeting standard objects. The state of a cobox consists of arbitrary many objects. This has several advantages. Often objects rely on other objects to hold additional state [NVP98, PS98, PHS07]. For dynamically growing state this is even mandatory. In pure active object approaches, additional state must be hold by other active objects. But this means that even in simple cases one has to deal with concurrency. Some approaches, like Creol [JOY06] for example, use functional data-types to hold additional state. To fully support standard object-oriented programming one must allow additional objects inside an active object. ASP [CHS04] supports such objects, called passive objects, which can only be accessed inside an active object. But only supporting complex internal state is not enough in the object-oriented setting. One also must allow for multiple service objects, i.e., multiple objects should be accessible from the outside. Otherwise it is not possible to support scalable interfaces consisting of arbitrary many objects, which often appear in OOP. The E programming language [MTS05], for example, supports this. The cobox model supports complex internal state as well as complex object interfaces. Note that this model is a generalization of approaches that do not support this because a cobox with a single object can be regarded as an active object in the traditional sense.

Most active object approaches have in common that there can only be one activity inside an active object. This is a restriction when an active object has to wait for
certain messages because meanwhile there cannot be any other activity in the object. In addition, it makes it difficult to support multiple different control flows, as these have to be encoded in a single control flow. Multiple activities inside an active object have been introduced in Creol [JO07] and is adopted by the coobox model. Especially in the combination with futures, multiple activities are appealing. In the coobox model these activities are called tasks. In contrast to a standard thread model, tasks are owned by the coobox and cannot “jump” to other cooboxes. Thus the overall behavior of a coobox is defined by the behavior of all its tasks. Tasks can thus be seen as parts of a coobox that define a sub-behavior. Tasks are scheduled cooperatively, which prevents data races and drastically reduces the number of possible interleavings compared to the preemptive setting.

In the original actor model [HBS73, Agh86], a script (or body) defines the behavior of an actor. In particular, it defines when certain messages are accepted. This allows the actor to only react to messages which it currently expects. Other messages are either discarded or ignored and treated later. A script can thus be used to define a fixed communication protocol. Languages that support such scripts typically have some kind of accept expression, which specifies by patterns, which messages are accepted. ASP [CHS04], for example, has a Serve expression, which specifies which methods can be served next. The coobox model does not support such a script, but it is possible to extend the model accordingly. The difficulty in the coobox model is that multiple service objects can be targets of messages. It is future work to investigate how to usefully specify possible next messages in the presence of such complex interfaces. In the coobox model messages are served in a FIFO order. It is possible to use monitor-like condition variables [Hoa74], which can be modeled by promises and futures (cf. Section 4.2.3), to realize state-dependent message handling.

To make it easier to compare the coobox model with other message-based programming models and languages, we give in Table 2.1 a comparison based on several aspects. We take only approaches into account that are based on the actor or active object approach, we do not consider approaches that are based on channel communication. Note that this is only an approximate comparison and makes no claim of being complete or precise. There are many details of the different approaches, which are not addressed, like the ordering of messages, for example. In addition, the concepts are often realized in slightly different ways.
### 2.5 Related Work and Discussion

<table>
<thead>
<tr>
<th>Model</th>
<th>Multiple Internal Objects</th>
<th>Multiple Service Objects</th>
<th>Asynchronous Method Calls</th>
<th>Explicit Message Acceptance</th>
<th>Multiple Local Control Flows</th>
<th>Futures</th>
<th>Synchronous Communication</th>
<th>Data Race Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCL/1 [YBS86]</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>○</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>POOL2 [Ame89]</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Hybrid [Nie87]</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>○</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>SALSA [VA01]</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ASP [CH05]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Creol [JO04]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JR [KGM04]</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>E [MTS05]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AmbientTalk/2 [VCMB+07]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Thorn [BFN+09]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CoBox Model</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**LEGEND**

- ✓ has the feature
- - does not have the feature
- ○ limited form of the feature

**Table 2.1: Comparison of different programming models**
Chapter 3
Core Calculus

This chapter presents a formal calculus for the cobox model, called JCoBox\textsuperscript{C}. The calculus serves as a precise definition of the semantics of the cobox programming model. The calculus is based on a Java-like core language with standard object-oriented features like classes, inheritance, method calls, and mutable state. On top of this language, the cobox model is realized by introducing cobox classes, asynchronous method calls, cooperative multitasking, futures, promises, and transfer classes. The calculus does not treat immutable objects as immutable objects can be regarded as restricted transfer objects, where object transfer is optimized for performance reasons.

Overview. Section 3.1 presents the abstract syntax of JCoBox\textsuperscript{C} and defines several auxiliary functions and predicates that are defined on the abstract syntax. The static semantics of JCoBox\textsuperscript{C} is given in Section 3.2. It precisely defines, which JCoBox\textsuperscript{C} programs are legal, by means of context conditions and a type system. Section 3.3 defines the dynamic semantics of JCoBox\textsuperscript{C} by a small-step, operational transition semantics. Type soundness of JCoBox\textsuperscript{C} is proven in Section 3.4, by proving the Preservation and Progress Lemmas. Additional properties of JCoBox\textsuperscript{C} are presented in Section 3.5, which include properties concerning data races, deadlocks, and determinism. Finally, Section 3.6 discusses JCoBox\textsuperscript{C} and relates it to other calculi for concurrent OOP.

3.1 Syntax

The abstract syntax of JCoBox\textsuperscript{C} is shown in Table 3.1. Before the syntax is explained in more detail, we briefly explain the notations that are used in the following.
\[
p \in \text{Prog} ::= D \ e
\]

\[
d \in D \subseteq \text{ClassDecl} ::= \mu \ \text{class} \ c \ \text{extends} \ c' \ \{ \ \tau f; \ H \}
\]

\[
\mu \in \text{Modifier} ::= \text{cobox} | \text{transfer} | \text{plain}
\]

\[
h \in H \subseteq \text{MethDecl} ::= \tau m(\tau x\{e\}
\]

\[
e \in \text{Expr} ::= \begin{align*}
&x \\
&\| \ \text{null} \\
&\| \ \text{let} \ x = e \ \text{in} \ e' \\
&\| \ e.f \\
&\| \ e.f = e' \\
&\| \ \text{new} \ c \ [\text{in} \ e] \\
&\| \ e.m(\bar{e}) \\
&\| \ e!m(\bar{e}) \\
&\| \ e.get \\
&\| \ e.await \\
&\| \ \text{yield} \\
&\| \ \text{promise} \ \tau \\
&\| \ e.fut \\
&\| \ e.resolve \ e'
\end{align*}
\]

\[
\tau \in \text{Type} ::= \begin{align*}
&c \\
&\| \ F(\tau) \\
&\| \ P(\tau)
\end{align*}
\]

\[
c \in \text{ClassName} \\
\]

\[
m \in \text{MethName} \\
\]

\[
f \in \text{FieldName} \\
\]

\[
x \in \text{VarName}
\]

Table 3.1: Abstract syntax of JCoBox\(^c\).
3.1 Syntax

Notations

We use the standard mechanism of meta-variables to represent elements of certain syntactic categories. For example, when using a variable with name $p$, it is implicitly assumed that $p \in \text{Prog}$, without explicitly mentioning it. We lift the idea of meta-variables to sets of elements, which are denoted by capital letters. For example, writing $D$ implicitly implies that $D \subseteq \text{ClassDecl}$. Analogously, an overbar, like $\bar{s}$, denotes sequences of elements. We use the standard notations for sets, i.e., $\cup$ for the union of sets and $\cup$ for the union of disjoint sets. We also assume the standard equality definition on sets, i.e., modulo reordering of elements. For sequences, the following notations are used. The empty sequence is denoted by $\cdot$; a centered dot, $\cdot$, adds an element to a sequence, which can be done from both sides; and $\circ$ concatenates two sequences. The length of a sequence is obtained by $\mid \_ \mid$. The predicate $\text{nodups} (\bar{s})$ is satisfied if sequence $\bar{s}$ contains no duplicates. Given a sequence $\bar{s}$, we write $s_i$ to select the $i$th element from the sequence and $s_n$ for selecting the last element of the sequence. Writing $\bar{s}[s_i = s]$ replaces the $i$th element of $\bar{s}$ with $s$. We often implicitly treat single elements as sequences or sets of size one when technically needed. For example, we write $S \cup s$ instead of $S \cup \{s\}$. In addition, functions defined on single elements are implicitly lifted to functions on sequences of elements. We use the wildcard _ to match an arbitrary term of the abstract syntax and the wildcard ... to match a sequence of terms. Terms enclosed by square brackets, [__], are optional.

3.1.1 Abstract Syntax

The abstract syntax of JCoBox is based on a Java-like syntax for the core object-oriented features.

Programs, Classes, and Methods. A program $p$ is a pair $D e$ consisting of set of class declarations $D$ and a main expression $e$. A class declaration $d$ consists of a class name $c$, a mandatory super class name $c'$, a sequence of field declarations $\tau f$, and a set of method declarations $H$. A class declaration can have a modifier $\mu$, which can either be cobox, to declare a cobox class, transfer to declare a transfer class, or plain to declare a standard plain class. We assume a predefined plain class Object with no fields, no methods and no super class. Method declarations consist of a return type, a method name, a sequence of parameters and a single body expression. All methods have a return value, which is the value of their body expression.

Expressions. The standard expressions of JCoBox include variables, the null constant, let expressions, field select and field update expressions.

Objects are created by the standard new $c$ expression. By default, an object is created in the current cobox, i.e., the cobox of the creating task. To create an object in a different cobox the extended new expression new $c$ in $e$ has to be used. The object is then created in the same cobox as the object that is referred by the evaluated target.
expression \( e \). If a class is declared as a cobox class, objects of that class are always created in a new cobox. For these classes, it is not possible to specify a target cobox. For example, if \( c \) is a cobox class then the expression \( \text{new } c \) creates an object of \( c \) in a new cobox. The \( \text{new } c \) expression is then evaluated to a reference to the new object.

Standard method calls, which we call direct calls in the following, are written \( e.m(\overline{e}) \). These calls are executed in a standard sequential stack-like way by the calling task. Targets of direct calls can only be near objects, i.e., they must belong to the cobox of the invoking task. Asynchronous method calls are indicated by an exclamation mark \( ! \) instead of a dot. They can target near and far objects and always result in a new task in the cobox of the target object. The result of an asynchronous method call is a reference to a future. Futures can be claimed exclusively using the get expression and cooperatively using the await expression. Explicit task scheduling is done by the yield expression. A promise for a value of type \( \tau \) is created by the promise \( \tau \) expression. To obtain a new future associated to a promise, the \( e.fut \) expression is used. The expression \( e.resolve \overline{e} \) resolves a promise \( e \) to the evaluated value of \( \overline{e} \). A type \( \tau \), can either be a class name \( c \), a future type \( F(\tau) \), or a promise type \( P(\tau) \).

### 3.1.2 Auxiliary Functions

#### Substitution

A variable \( x \) is free in an expression \( e \) if it is not bound by a \( \text{let} \)-expression. Replacing all occurrences of the free variable \( x \) in expression \( e \) with expression \( \overline{e} \), is done by a standard capture-avoiding substitution, denoted by \( [\overline{e}/x]e \). The precise definition is given in Definition A.1 on Page 163.

#### Functions on the Abstract Syntax

Table 3.2 gives several functions and predicates that extract information from the abstract syntax tree of programs. Most functions and predicates are parametrized with a subscript to indicate the context program \( p \). In most cases, this subscript is left out in the following as we always assume a fixed program \( p \). For simple selector functions, we often use the meta-variable name of the corresponding term as function name, but in a non-italic font. For example, the function \( c_d \) selects the class name \( c \) from class declaration \( d \).

**Remark.** As in Table 3.2, we often use rules to define functions. Strictly speaking this is formally not correct as, for example, a rule that derives \( \text{fun}(x) = y \), does not define a function, but rather defines that a certain syntactic term can be derived, where, in particular, the equal sign \( = \) has no special meaning. In our case, if a rule derives a term with an equal sign, we take the equal sign with standard equality axioms namely reflexivity, commutativity, and transitivity, without making this explicit in the rules. Note that, in addition, the rules often do not define functions, but actually
3.1 Syntax

\[ p = D \quad d \in D_p \quad d = \_ \text{class} \ c \ldots \quad \text{decl}_p(c) = d \quad \text{defined}_p(c) \quad \text{decl}_p(c) = \mu \text{class} \ c \ldots \quad \text{modifier}_p(c) = \mu \]

\[
\begin{align*}
\text{modifier}_p(c) &= \text{cobox} \\
\text{coboxcl}_p(c) &= \text{transfer} \\
\text{transfercl}_p(c) &= \text{plain} \\
\text{plaincl}_p(c) &= \mu
\end{align*}
\]

\[
\begin{align*}
d &= \ldots \text{class} \ c \ldots \\
\text{c}_d &= c \\
\text{supercl}(\text{decl}_p(c)) &= c' \\
\text{supercl}_p(c) &= c'
\end{align*}
\]

\[
\begin{align*}
\text{methods}_p(\text{Object}) &= \emptyset \\
\text{methods}_p(c) &= H \\
\text{decl}_p(c) &= \_ \text{class} \ c \text{ extends } c' \{ \_; H \} \\
h &= \_m(\_\{\_\} \\
m_h &= m
\end{align*}
\]

\[
\begin{align*}
h &\in \text{methods}_p(c) \\
m_h &= m \\
\text{mdecl}_p(c,m) &= h \\
h &\notin \text{methods}_p(c) \\
\text{supercl}_p(c) &= c' \\
\text{supercl}_p(c', m) &= h \\
\text{mdecl}_p(c,m) &= h \\
\text{mdecl}_p(c,m) &= \tau \\n\text{mbody}_p(c,m) &= \bar{x}.e \\
\text{mtype}_p(c,m) &= \tau \\
\text{mbody}_p(c,m) &= \bar{x}.e \\
\text{mexpr}_p(c,m,r,v) &= [r/\text{this}, v/\bar{x}]e
\end{align*}
\]

\[
\begin{align*}
\text{decl}_p(c) &= \_ \text{class} \ c \text{ extends } c' \{ \_ \tau f ; \_ \} \\
\text{fields}_p(\text{Object}) &= \bullet \\
\text{fields}_p(c) &= \tau f \circ \text{fields}_p(c')
\end{align*}
\]

Table 3.2: Functions and predicates on the abstract syntax

Define relations. For example, in Table 3.2, there may exist several class declarations with the same name, so that decl is a relation and not a function. However, for well-typed programs (see Section 3.2.3), all given relations are actually (partial) functions, which can be proven straightforwardly.
3.2 Static Semantics

This section presents the type system for JCoBox\textsuperscript{c}. The type system itself is a straightforward modification of type systems known from other formalizations of Java-like languages like Featherweight Java [IPW01] or CLASSICJAVA [FKF99]. The main differences are that it additionally covers asynchronous method calls, futures, and promises.

3.2.1 Subtyping

The subtyping relation is the reflexive and transitive closure of the subtyping induced by the class declarations and by the future subtyping rule (see Table 3.3). In addition, future types are covariant in their type parameter.

\[
\begin{align*}
\text{(S-DEF)} \quad \text{supercl}_p(c) &= c' \\
\text{c} <:_p c' \\
\text{(S-REFL)} \quad \tau <:_p \tau \\
\text{(S-TRANS)} \quad \tau <: p \tau' & \quad \tau' <: p \tau'' \\
\text{f}(\tau) <: p f(\tau') \\
\text{(S-FUT)} \quad \tau <: p \tau'
\end{align*}
\]

Table 3.3: Subtyping rules of JCoBox\textsuperscript{c}

3.2.2 Context Conditions

JCoBox\textsuperscript{c} has context conditions known from Java and has some additional restrictions (see Table 3.4):

1. The set of class declarations must have unique names (C-UNIQUENAMES).
2. A class must not extend itself (C-NOSELFEXTEND).
3. The subtyping relation is antisymmetric and thus a partial order (C-NOCYCLES).
4. All fields of a class, including the fields of its super classes must have unique names (C-NOFIELD_HIDE).
5. Methods must not be overloaded (C-NOOVERLOAD).
6. Methods must be correctly overridden (C-OVERRIDE_OK).

JCoBox\textsuperscript{c} has stricter conditions than Java as JCoBox\textsuperscript{c} allows neither field hiding nor method overloading. In addition, neither contravariant parameter types nor covariant return types are allowed when overriding methods.
3.2 Static Semantics

∀d, d′ ∈ D . c_d = c_d′ ⇒ d = d′

uniqueNames(D)

∀d, d′ ∈ D . c_d <p c_d′ ∧ c_d′ <p c_d ⇒ d′ = d

noCycles_p(D)

(c-NoSelfExtend)

noSelfExtend(c, c′)

c ≠ c′

(c-FieldHide)

nodups(fields_p(c))

(c-NoOverload)

noOverload(H)

∀h, h′ ∈ H . m_h = m_h′ ⇒ h = h′

(c-Overlap)

∀h, h′ ∈ H . m_h = m_h′ ⇒ h = h′

(c-NoCycles)

∀d, d′ ∈ D . c_d = c_d′ ⇒ d = d′

(c-UniqueNames)

∀d, d′ ∈ D . c_d = c_d′ ⇒ d = d′

Table 3.4: Context conditions for JCoBox\(^c\) programs.

3.2.3 Typing Rules

The judgments of the type system are shown in Table 3.5. The typing rules of JCoBox\(^c\) are shown in Table 3.7. We assume that all typing rules are evaluated in the context of a fixed program \(p\). This fixed program is thus an implicit parameter of all type judgments. We sometimes lift the type judgments from single elements to sets of elements. Lifted type judgments are indicated by a \(*\). For example, the judgment \(p \vdash_\theta \emptyset\) means that \(p \vdash_\theta d\) holds for all \(d \in D\). Sequences are treated in an analogous way.

| \(\vdash_p p\) | Program \(p\) is well-typed |
| \(\vdash_d d\) | Class declaration \(d\) is well-typed |
| \(c \vdash_h h\) | Method declaration \(h\) is well-typed under class name \(c\) |
| \(\Theta \vdash_e e : \tau\) | Expression \(e\) has type \(\tau\) under \(\Theta\) |
| \(\Theta \vdash_e e :< \tau\) | Expression \(e\) can be typed to a subtype of \(\tau\) under \(\Theta\) |
| \(\vdash_\tau\) | Type \(\tau\) is a valid type. |

Table 3.5: Type judgments of the static semantics.

\(^1\)Note that this implies that an empty set is always well-typed
Type Environments

Expressions are typed under a program $p$ and a type environment $\Theta$ (cf. Table 3.6). $\Theta$ is a pair, $\Gamma; \Sigma$, consisting of a variable typing $\Gamma$ and a reference typing $\Sigma$. The variable typing $\Gamma$ assigns types to variables, the reference typing assigns types to references $\Sigma$. $\Sigma$ is technically needed for the type soundness proof to type object references (cf. Section 3.4), which can appear in expressions during the evaluation of JCoBox$^C$ programs. $\Sigma$ is not needed to type source expressions, i.e., expressions that can be used to write JCoBox$^C$ programs. All source expressions are typed under an empty reference typing $\emptyset$. We write $\Gamma, x : \tau$ to extend $\Gamma$ with an additional map from $x$ to $\tau$, and $\Sigma, r : \tau$, to extend $\Sigma$, respectively. We sometimes use a functional-like syntax to obtain the type from a typing, e.g. $\Gamma(x) = \tau$.

\[
\begin{align*}
\Theta &::= \Gamma; \Sigma & \text{type environment} \\
\Gamma &::= \emptyset | \Gamma, x : \tau & \text{variable typing} \\
\Sigma &::= \emptyset | \Sigma, r : \tau & \text{reference typing}
\end{align*}
\]

Table 3.6: Type environments

Subsumption

For a more concise notation we use a subsumption rule (T-SUB), written $\Theta \vdash_e e :< \tau$, which simply means that $e$ is typed to some subtype of $\tau$. We use the slightly different syntax $:<$ to make clear when this rule is applied. In proofs we often implicitly use the inversion of the rule, i.e.,

$$\Theta \vdash_e e :< \tau \Rightarrow \exists \tau' \cdot \tau' <: \tau \land \Theta \vdash_e e : \tau',$$

where we are often only interested in the fact that $\Theta \vdash_e e : \tau'$ for some type $\tau'$.

Programs, Classes, and Methods

Programs (T-PROGRAM). A program, $p = D e$, is correctly typed if its class declarations $D$ are correctly typed, and its main expression $e$ is correctly typed under the empty type environment. In addition, the set of class declarations must neither contain duplicate class names nor subtyping cycles.

Classes (T-DECL). A class declaration $d$ with class name $c$ is well-typed if all its method declarations are well-typed under $c$. In addition, several context conditions must hold as described in Section 3.2.2.
3.2 Static Semantics

\[
\begin{array}{c}
(T\text{-}Program) \\
p = D e \quad uniqueNames(D) \\
\text{noCycles}(D) \quad \vdash_d^* D \quad \emptyset; \emptyset \vdash_e e \\
\vdash_p p \\
\end{array}
\]

\[
(T\text{-}Decl) \\
\text{noSelfExtend}(c, c') \quad \text{noFieldHide}(c) \\
\text{noOverload}(H) \quad \text{modifierOk}(c, c') \quad c \vdash_h^* H \quad \vdash_i^* \tau \\
\vdash_d \text{ class } c \text{ extends } c'\{ \overline{\tau f, H} \}
\]

\[
(T\text{-}Method) \\
h = \tau m(\overline{x}{e}) \\
nodups(this \cdot x') \quad \text{overrideOk}(c, m) \quad \text{this : } c, x : \overline{\tau}; \emptyset \vdash_e e : < \tau \\
c \vdash_h h
\]

\[
(T\text{-}Null) \quad (T\text{-}Var) \quad (T\text{-}Let) \\
\vdash \tau \quad \Gamma(x) = \tau \quad \Gamma; \Sigma \vdash_e x : \tau \quad \Gamma, x : \tau; \Sigma \vdash_e e' : \tau' \\
\emptyset \vdash_e \text{null : } \tau \quad \Gamma; \Sigma \vdash_e x : \tau \\
\vdash \text{new } c : c \quad \vdash \text{new } c : c
\]

\[
(T\text{-}New) \quad (T\text{-}NewIn) \quad (T\text{-}FieldSelect) \\
\vdash_1 c \quad \emptyset \vdash_e e : c' \quad \text{plaincl}(c) \\
\emptyset \vdash_e \text{new } c : c \quad \emptyset \vdash_e \text{new } c : c \quad \emptyset \vdash_e e : c \\
\emptyset \vdash_e e.f_i : \tau_i
\]

\[
(T\text{-}FieldUpdate) \quad (T\text{-}Yield) \quad (T\text{-}FutAwait) \quad (T\text{-}FutGet) \\
\emptyset \vdash_e e.f : \tau \quad \emptyset \vdash_e e' : < \tau \\
\emptyset \vdash_e \text{yield : } \tau \quad \emptyset \vdash_e \text{wait : } \tau \quad \emptyset \vdash_e e : f(\tau)
\]

\[
(T\text{-}PromNew) \quad (T\text{-}PromResolve) \quad (T\text{-}PromFut) \\
\vdash_1 \tau \\
\emptyset \vdash_e \text{promise } \tau : p(\tau) \\
\emptyset \vdash_e e : p(\tau)
\]

\[
(T\text{-}DirectCall) \quad (T\text{-}AsyncCall) \quad (T\text{-}Sub) \\
\emptyset \vdash_e e : c \quad \emptyset \vdash_e \overline{\overline{c}} : < \overline{\tau} \\
\text{mtype}(c, m) = \overline{\tau \triangleright \tau} \\
\emptyset \vdash_e e.m(\overline{e}) : \tau \\
\emptyset \vdash_e e.a.m(\overline{e}) : f(\tau) \\
\emptyset \vdash_e e : < \tau
\]

\[
(T\text{-}TypeCl) \quad (T\text{-}TypeFut) \quad (T\text{-}TypeProm) \\
definedcl(c) \\
\vdash_1 c \\
\vdash_1 f(\tau) \\
\vdash_1 p(\tau)
\]

Table 3.7: Expression typing of JCoBox
Chapter 3  Core Calculus

Methods (T-METH). A method declaration $h$ is correctly typed if (1) the names of the formal parameters, including this, do not contain duplicates; (2) it does not incorrectly override a method; (3) its body expression is typed to a subtype of the return type of the method, under a variable typing that maps the formal parameter names to their corresponding declared types, including the implicit this parameter.

Expressions

Expressions are typed as follows. The null constant can be typed to any type. The type of a let-expression is the type of its second expression, where a typed variable with the type of the first expression is substituted for the bound variable (T-LET). A new-expression is typed to its class name (T-NEW). The extended new expression, new $c$ in $e$, additionally requires that $e$ is typed to a class and that $c$ is a plain class. The reason for the latter requirement is that transfer objects cannot be created in a different cobox, and instances of cobox classes are always created in a new cobox and thus specifying a target cobox makes no sense.

A field select requires that the target expression is typed to a class, and the selected field exists in the field declarations of that class. A field select is then typed to the type of the selected field (T-FIELDSELECT). A field update in addition requires that the type of the right-hand expression is a subtype of the field’s type (T-FIELDUPDATE).

A yield-expression can be typed to any valid type. The reason is that yield is reduced to null by the operational semantics. This way the introduction of a unit type is avoided. The expressions await and get require that their subexpressions are typed to a future type. They are then typed to the type parameter of that type (T-FUTWAIT) and (T-FUTGET).

Method invocations ((T-DIRECTCALL) and (T-ASYNCCALL)) require that the types of the argument expressions are subtypes of the formal argument types and that the type of the body expression is a subtype of the method’s return type. The invocations are then typed to the declared return type of the called method, or, in the asynchronous case, to the corresponding future type.

3.3 Dynamic Semantics

This section presents the dynamic semantics of JCoBox$^{\text{c}}$. The formalization given in this section is also defined in Maude [CDE+07]. The Maude formalization is presented in Appendix B.

3.3.1 Evaluation Contexts

To abstract from the context of an expression and to define the evaluation order of expressions we use evaluation contexts [FH92]. An evaluation context is an expression with a “hole” $\Box$ at a certain position. By writing $e[\Box]\{e\}$ that hole is replaced by the expression $e$. Table 3.8 defines the possible contexts.
### 3.3 Dynamic Semantics

\[ e_{\square} ::= \square | e_{\square}.f | e_{\square}.f = e | v.f = e_{\square} | \text{new } c \text{ in } e_{\square} \\
| \text{let } x = e_{\square} \text{ in } e_{\square}.n(\overline{e}) \text{ in } v.n(\overline{e}, e_{\square}, \overline{e}) \text{ in } e_{\square}.n(\overline{v}, e_{\square}, \overline{e}) \\
| e_{\square}.\text{get} | e_{\square}.\text{await} | e_{\square}.\text{fut} | e_{\square}.\text{resolve } e | v.\text{resolve } e_{\square} \]

#### Table 3.8: Evaluation contexts

| Config ::= K | configurations |
| k ∈ K ⊆ Comp ::= b | ρ | components |
| b ∈ B ⊆ CoBox ::= b(κ_b, O, T, \overline{t}) | coboxes |
| ρ ∈ P ⊆ Prom ::= p(κ_p, O, v_e) | promises |
| o ∈ O ⊆ Obj ::= o(t, c, \overline{v}) | objects |
| t ∈ T ⊆ Tsk ::= \tau(e) | futures |
| v ⊆ Value ::= r | null | tasks |
| v_e ⊆ OptValue ::= v | e | values |
| r ∈ R ⊆ Ref ::= \kappa.t | \kappa_p | optional values |
| κ ∈ CompId ::= κ_b | κ_p | references |
| c ∈ CoBoxId ::= κ_b | κ_p | component identifiers |
| v ⊆ Expr ::= . . . | v | extended expressions |
| t ∈ ObjId | object identifiers |
| κ_b ∈ CoBoxId | cobox identifiers including \text{MAIN} |
| κ_p ∈ PromId | promise identifiers |

#### Table 3.9: Semantic entities of JCoBox\(^c\). Small capital letters like \(b\) or \(o\) are used as “constructors” to distinguish the different semantic entities syntactically.

### 3.3.2 Semantic Entities

The semantic entities used by the semantics are shown in Table 3.9.

**Configurations.** The state of a program is represented by a configuration \(K\), which is a set of components \(k\). A component can either be a cobox \(b\) or a promise \(ρ\).

**CoBoxes.** A cobox, \(b(κ_b, O, T, \overline{t})\), consists of a globally unique cobox identifier \(κ_b\), a set of objects \(O\), a set of suspended tasks \(T\) and a sequence of tasks \(\overline{t}\). If \(\overline{t} = \overline{t}' \cdot t\), \(t\) represents the active task and \(\overline{t}'\) represents the ready queue of the cobox. Given a cobox \(b = b(κ_b, O, T, \overline{t})\), we define the following selector functions: \(\text{id}_b \doteq κ_b\), \(O_b \doteq O\), and \(T_b \doteq T\).
Promises. Promises, $r(p, O, v_e)$, are “degenerated” coboxes that do not have tasks. Instead, a promise has an optional value $v_e$, which is $e$ as long as the promise is not resolved. Like a cobox, a promise also has a set of objects $O$. This set is initially empty. When the promise is resolved to a value $v$, it contains all transfer objects and futures, reachable by $v$ via transfer objects. Like on coboxes, we define selector functions on promises. Let $\rho = r(p, O, v_e)$, then \(\text{id}_\rho \triangleq p\), \(\text{O}_\rho \triangleq O\), and \(v_\rho \triangleq v_e\).

Objects. Objects, $o(\iota, c, \nu)$, consist of an object identifier $\iota$, a class name $c$, and a sequence of values $\nu$ representing its state. The object identifier is only unique in the context of a single component. Objects of different components may have the same identifier. An object always belongs to a certain component and cannot move to a different one. Let $\omega = o(\iota, c, \nu)$, then \(\text{id}_\omega \triangleq \iota\), and \(c_\omega \triangleq c\).

Futures. Futures, $F(\iota, p, v_e)$, are similar to objects, but always have a reference to its associated promise $p$ and have an optional value $v_e$, which is $e$ until the future is resolved. Let $\omega = F(\iota, p, v_e)$, then \(\text{id}_\omega \triangleq \iota\), \(\text{pid}_\omega \triangleq p\), and \(v_\omega \triangleq v_e\).

Tasks. A task $T(e)$ only consists of a single expression $e$, which is in general of the form $\kappa_p \cdot \text{resolve } e'$, where $\kappa_p$ is the promise that is resolved by the task. Only the initial task of a program has no associated promise. Tasks that are suspended, i.e., belong to the suspend set of a cobox, always have the form $e \cdot \text{get}$, i.e., wait for a future.

Values. Values are either references $r$ or null, where a reference can either be an object reference $\kappa.\iota$ or a promise identifier $\kappa_p$.

Object References. An object reference consists of the identifier $\kappa$ of the component the object belongs to, and a component-local identifier $\iota$. By defining references that way, it is possible to locally create objects with globally unique references. Given the identifier $\kappa$ of the context component, a reference $\kappa'.\iota$ is near if $\kappa' = \kappa$ and far if $\kappa \neq \kappa'$.

3.3.3 Auxiliary Functions

Table 3.10 shows the definition of auxiliary functions used by the semantics. The function \(\text{oids}(O)\) extracts the set of object identifiers from $O$ including the identifiers of futures. \(\text{init}(c)\) returns a sequence of null values, whose length is equal to the number of fields of $c$, and is used to initialize the attributes of a newly created object.

Data Transfer. Transfer objects and futures are copied between components by the \textit{copy} function. The \textit{copy} function uses the \(\text{reach}(O, \nu)\) function to extract all transfer objects of $O$ transitively reachable by other transfer objects starting with $\nu$. In addition,
3.3 Dynamic Semantics

oids\( (O) \triangleq \{ o | o(t, _, _) \in O \lor f(t, _, _) \in O \} \)

init\( (c) \triangleq \text{null} \) where |null| = |fields\( (c) |

reach\( (\emptyset, \emptyset) \triangleq \emptyset \)

reach\((O, \bullet) \triangleq \emptyset \)

reach\((O \cup o, \nu \cdot \kappa \cdot t) \triangleq \text{reach\((O, \nu \circ \nu') \cup \{ o \}) \) if o = o(t, c, \nu') \land \text{transfer\(\ell\( (c) \)

reach\((O \cup o, \nu \cdot \kappa \cdot t) \triangleq \text{reach\((O, \nu) \cup \{ f(t, \kappa_p, \epsilon) \} \) if o = f(t, \kappa_p, \nu_\epsilon) \)

reach\((O \cup o, \nu \cdot v) \triangleq \text{reach\((O \cup o, \nu) \) else \)

copy\((\kappa, O, \nu, \kappa', O') \triangleq (\sigma O'', \sigma \nu) \)

where \(O'' = \text{reach\((O, \nu) \)

and \(\sigma = \{ \kappa \cdot t \mapsto \kappa' \cdot t', t \mapsto t' | t \in \text{oids\((O'') \land t' \text{ fresh} \} \)

Table 3.10: Auxiliary functions

Futures are extracted, but their value is reset to \(\epsilon\) and not regarded for the reachability of objects. This point is important for a deterministic transfer of futures, independent of their actual resolving status. Whenever a future is transferred to another cobox it has to be resolved again by the associated promise. The function \(\text{copy}\((\kappa, O, \nu, \kappa', O')\) creates a copy of all objects from component \(\kappa\), which are identified by the reach\(\) function. The copied objects get fresh identifiers for component \(\kappa'\). The set \(O\) represents the objects of \(\kappa\), where \(O'\) represents the objects of \(\kappa'\).

3.3.4 Transition Rules

The dynamic semantics of JCoBox\(^c\) is defined as a small-step, operational, transition semantics, which is given by the relation \(K \longrightarrow K'\). The relation is implicitly parametrized by a fixed underlying program, which we omit for conciseness. The rules defining the relation are split into two parts: cobox-local rules (cf. Table 3.11), which are only defined on the state of coboxes, and global rules (cf. Table 3.12), which require the complete configuration state. Splitting up the rules in this way makes it explicit which steps can be executed in isolation and which require interaction between components.

CoBox-Local Rules

The relation for cobox-local rules is denoted by \(\longrightarrow_b\) and is defined on coboxes (see Table 3.11). These rules essentially model standard sequential programming inside a cobox, together with cooperative multitasking.
Chapter 3 Core Calculus

(R-LET)
\[ b(\kappa, O, T, \bar{t} \cdot \tau(e_\square[\text{let } x = v \text{ in } e])) \rightarrow_b b(\kappa, O, T, \bar{t} \cdot \tau(e_\square[x/v]e)) \]

(R-NEWLOCAL)
\[ e = \text{new } c \lor e = \text{new } c \text{ in } \kappa, t' \quad \neg \text{coboxcl}(c) \quad t \notin \text{oids}(O) \]
\[ b(\kappa, O, T, \bar{t} \cdot \tau(e_\square[e])) \rightarrow_b b(\kappa, O \cup o(\langle t, c, \text{init}(c) \rangle), T, \bar{t} \cdot \tau(e_\square[\kappa, t])) \]

(R-DIRECTCALL)
\[ o(\langle t, c, _ \rangle) \in O \quad e' = \text{mexpr}(c, m, \kappa, t, \vec{v}) \]
\[ b(\kappa, O, T, \bar{t} \cdot \tau(e_\square[\kappa, t, m(\vec{v})])) \rightarrow_b b(\kappa, O, T, \bar{t} \cdot \tau(e_\square[e'])) \]

(R-FIELDSELECT)
\[ o(\langle t, c, \vec{v} \rangle) \in O \quad \text{fields}(c) = \tau \bar{f} \]
\[ b(\kappa, O, T, \bar{t} \cdot \tau(e_\square[\kappa, t, f_i])) \rightarrow_b b(\kappa, O, T, \bar{t} \cdot \tau(e_\square[v_i])) \]

(R-FIELDUPDATE)
\[ o = o(\langle t, c, \vec{v} \rangle) \quad \text{fields}(c) = \tau \bar{f} \quad o' = o(\langle t, c, \vec{v}[v_i = v] \rangle) \]
\[ b(\kappa, O \cup o, T, \bar{t} \cdot \tau(e_\square[\kappa, t, f_i])) \rightarrow_b b(\kappa, O \cup o', T, \bar{t} \cdot \tau(e_\square[v_i])) \]

(R-FUTGET)
\[ f(\langle t, _ \rangle, \vec{v}) \in O \]
\[ b(\kappa, O, T, \bar{t} \cdot \tau(e_\square[\kappa, t, \text{get}])) \rightarrow_b b(\kappa, O, T, \bar{t} \cdot \tau(e_\square[v])) \]

(R-FUTAwait)
\[ b(\kappa, O, T, \bar{t} \cdot \tau(e_\square[r, \text{await}])) \rightarrow_b b(\kappa, O, T \cup \tau(e_\square[r, \text{get}]), \bar{t}) \]

(R-TASKRESUME)
\[ f(\langle t, _ \rangle, \vec{v}) \in O \]
\[ b(\kappa, O, T \cup \tau(e_\square[\kappa, t, \text{get}]), \bar{t}) \rightarrow_b b(\kappa, O, T, \tau(e_\square[v]) \cdot \bar{t}) \]

(R-YIELD)
\[ b(\kappa, O, T, \bar{t} \cdot \tau(e_\square[yield])) \rightarrow_b b(\kappa, O, T, \tau(e_\square[\text{null}]) \cdot \bar{t}) \]

(R-TASKTERMINATE)
\[ b(\kappa, O, T, \bar{t} \cdot \tau(v)) \rightarrow b(\kappa, O, T, \bar{t}) \]

Table 3.11: JCoBox\textsuperscript{c} cobox-local rules. Important terms are emphasized with a highlighted background.
**Sequential Programming.** The sequential programming rules are more or less standard. The important aspect is that the target of field reads, updates, and direct method calls must be objects of the same cobox. In addition, object creation can only happen inside the current cobox.

**Future Operations.** If a future is claimed using `get`, the active task cannot proceed until the corresponding future is resolved. Such a task prevents any other task in its cobox to become active. When the future is resolved, the `get` expression is reduced to the value of the future (R-FutGet). When a future is claimed by using `await`, the active task is moved to the suspend set $T$ (R-FutAwait), which also means that the next task of the task queue becomes active. In addition, the `await` expression is replaced with `get`. A suspended task is resumed when the future that the task is waiting for is resolved (R-TskResume). A resumed task is moved to the end of the task queue, and the `get` expression is replaced with the future value.

**Task Yielding and Termination.** The active task gives up control to the next ready task by the rule (R-Yield), which moves the active task $t$ to the end of the task queue. If there is no other task in the task queue, task $t$ stays active. A task terminates when its expression is reduced to a value (R-TskTerminate). Terminated tasks are simply removed from the task queue. Note that in general the expression $e$ of a task has the form $r$.resolve $e$. This means that, before $e$ is reduced to a value, the corresponding promise $r$ has already been resolved, which is done by global rule (R-PromResolve).

**Global Rules**

The global rules are defined on configurations (cf. Table 3.12). Each rule either requires more than one component, or creates a new component.

**Far Object and CoBox Creation.** Objects in a different cobox are created by rule (R-NewObjFar). It is essentially equal to (R-NewObjLocal), but creates the new object in a different cobox and requires a plain class. An object is created in a new cobox by the rule (R-NewCoBox). It requires that the parameter class is a cobox class. It then creates the object in a new cobox. The new cobox only consists of the new object – it has no tasks. The result of the new-expression is a reference to the newly created object.

**Promises.** Promises are created by the rule (R-PromNew). A new promise is initially unresolved and has an empty object heap. An unresolved promise is resolved by copying the value and all transitively reachable transfer objects and futures to the promise (R-PromResolve). An already resolved promise cannot be resolved again.
Chapter 3 Core Calculus

\[(\text{R-NewObjFar})\]

\[
plaincl(c) \quad b = b(\kappa_b', O', T', t') \quad t \notin oids(O') \quad b' = b(\kappa_b', O' \cup \circ(t, c, init(c)), T', \bar{t}')
\]

\[
K \cup b \cup b(\kappa_b, O, T, \bar{t} \cdot \tau(e_\square[\text{new } c \text{ in } \kappa_b', t'])) \rightarrow K \cup b' \cup b(\kappa_b, O, T, \bar{t} \cdot \tau(e_\square[\kappa_b', t']))
\]

\[(\text{R-NewCoBox})\]

\[
coboxcl(c) \quad \kappa_b' \text{ fresh} \quad b = b(\kappa_b', \{\circ(t, c, init(c))\}, \emptyset, \bullet)
\]

\[
K \cup b(\kappa_b, O, T, \bar{t} \cdot \tau(e_\square[\text{new } c])) \rightarrow K \cup b(\kappa_b, O, T, \bar{t} \cdot \tau(e_\square[\kappa_b', t'])) \cup b
\]

\[(\text{R-PromNew})\]

\[
\kappa_p \text{ fresh}
\]

\[
K \cup b(\kappa_b, O, T, \bar{t} \cdot \tau(e_\square[\text{promise } t])) \rightarrow K \cup b(\kappa_b, O, T, \bar{t} \cdot \tau(e_\square[\kappa_p])) \cup p(\kappa_p, \emptyset, e)
\]

\[(\text{R-PromResolve})\]

\[
\rho = p(\kappa_p, \emptyset, e) \quad (O', v') = \text{copy}(\kappa_b, O, v, \kappa_p, \emptyset) \quad \rho' = p(\kappa_p, O', v')
\]

\[
K \cup \rho \cup b(\kappa_b, O, T, \bar{t} \cdot \tau(e_\square[\kappa_p, \text{resolve } v])) \rightarrow K \cup \rho' \cup b(\kappa_b, O, T, \bar{t} \cdot \tau(e_\square[\text{null}]))
\]

\[(\text{R-PromFut})\]

\[
p(\kappa_p, _, _, \) \in K \quad t \notin oids(O)
\]

\[
K \cup b(\kappa_b, O, T, \bar{t} \cdot \tau(e_\square[\kappa_p, \text{fut}])) \rightarrow K \cup b(\kappa_b, O \cup \circ(t, \kappa_p, e), T, \bar{t} \cdot \tau(e_\square[\kappa_b', t']))
\]

\[(\text{R-FutResolve})\]

\[
O = O' \cup \circ(t, \kappa_p, e) \quad p(\kappa_p, O'', v) \in K \quad (O''', v') = \text{copy}(\kappa_p, O'', v, \kappa_b, O)
\]

\[
K \cup b(\kappa_b, O, T, \bar{t}) \rightarrow K \cup b(\kappa_b, O''', O' \cup \circ(t, \kappa_p, v'), T, \bar{t})
\]

\[(\text{R-AsyncCallLocal})\]

\[
e = mexpr(c, \kappa_b, t, m, \bar{v}) \quad t' = \tau(\kappa_p, \text{resolve } e) \quad \kappa_p \text{ fresh} \quad \rho = p(\kappa_p, \emptyset, e)
\]

\[
K \cup b(\kappa_b, O, T, \bar{t} \cdot \tau(e_\square[\kappa_b', t \cdot m(\bar{v})])) \rightarrow K \cup b' \cup b(\kappa_b, O, T, \bar{t} \cdot \tau(e_\square[\kappa_p, \text{fut}])) \cup \rho
\]

\[(\text{R-AsyncCallFar})\]

\[
\circ(t, c, _) \in O' \quad (O''', \bar{v}') = \text{copy}(\kappa_b, O, \bar{v}, \kappa_b', O') \quad e = mexpr(c, \kappa_b', t, m, \bar{v}')
\]

\[
t' = \tau(\kappa_p, \text{resolve } e) \quad b = b(\kappa_b', O', T', \bar{t}') \quad b' = b(\kappa_b', O' \cup O''', T', \bar{t}' \cdot \bar{t}') \quad \kappa_p \text{ fresh} \quad \rho = p(\kappa_p, \emptyset, e)
\]

\[
K \cup b \cup b(\kappa_b, O, T, \bar{t} \cdot \tau(e_\square[\kappa_b', t \cdot m(\bar{v})])) \rightarrow K \cup b' \cup b(\kappa_b, O, T, \bar{t} \cdot \tau(e_\square[\kappa_p, \text{fut}])) \cup \rho
\]

\[(\text{R-Congruence})\]

\[
b \longrightarrow_{\tau} b'
\]

\[
K \cup b \longrightarrow K \cup b'
\]

Table 3.12: JCoBox$^c$ global rules. Important terms are emphasized with a highlighted background.
Futures. Futures are created by obtaining a future from a promise by using the `fut` operation (R-PROMFUT). The new future is initially unresolved and is added to the local object heap of the current cobox. A future can be resolved at any time when its associated promise is resolved (R-FUTRESOLVE). All objects of the promise are then copied to the cobox of the future and the value of the future is set to the copy of the promise value.

Note that this means that the result of an asynchronous method call is essentially copied twice: first to the promise and then to the cobox of the future. This is needed, as futures can be passed to other coboxes and require to get a copy of the original result value stored in the promise. In practice, this double copying can be avoided in many cases, where it is statically clear that the future is not passed to another cobox.

Asynchronous Calls. Asynchronous method calls are distinguished into local (R-ASYNCALLLOCAL) and far calls (R-ASYNCALLFAR). The local one addresses the current cobox and does not copy the method parameters. The far one addresses a different cobox and copies the parameters. Both calls create a new promise for holding the result of the call and add a new task, which executes the body expression of the corresponding method, to the end of the task queue. Both call expressions are reduced to a $\kappa_p$ fut expression to obtain a future from the new promise.

Congruence. Finally, rule (R-CONGRUENCE) integrates the cobox-local relation into the global relation.

Initial Configuration

A JCoBox$^c$ program $p = D e$ is started by executing $e$ in the initial cobox of $p$:

$$b^p_{\text{init}} \triangleq b(\text{MAIN}, \emptyset, \emptyset, \tau(e))$$

This means that the initial configuration always consists of a cobox, even if there exists no cobox class in the program.

3.4 Type Soundness

Having defined the static and dynamic semantics of JCoBox$^c$, we now show its type soundness. Type soundness is a property which states that well-typed programs never go wrong. For example, in object-oriented languages this means that all method calls at runtime are understood by an object, i.e., the corresponding method is defined by its class.

The type system of JCoBox$^c$ guarantees that no configuration can exist at runtime where:

- A reference is used, for which no object exists.
• A field is read or updated on an object which does not have that field.
• A method is called on an object which does not have that method, or the number
  or type of the actual parameters does not match the formal parameters of that
  method.
• The operations `await` and `get` are applied to a non-future reference.
• The operations `get` and `resolve` are applied to a non-promise reference.

However, there are also some erroneous configurations, which can not be prevented
by the JCoBox^c type system, namely:

1. Reading or updating a field, or calling a method on `null`.
2. Executing `await`, `get`, `fut`, or `resolve` on `null`.
3. Trying to resolve an already resolved promise.
4. Reading or updating a field of a far object.
5. Directly calling a method on a far object.

All these errors can be handled by a richer type system. The first two cases can be
addressed by introducing non-null types [FL03], the third case can be addressed
by using a linear type system [ÁGGS09], finally the fourth and fifth cases can be
addressed by a simple form of an ownership type system similar to that of Clarke et
al. [CWOJ08].

Type soundness is shown by two lemmas: Progress and Preservation [WF94, Pie02].
Progress in our setting means that a well-typed configuration is either a terminal
configuration (including stuck and erroneous configurations) or it can make a step in
the operational semantics. Preservation means that if a well-typed configuration can
do step, the next configuration is again well-typed.

### 3.4.1 Typing of Semantic Entities

For the type soundness proof we have to define the typing of semantic entities.
Table 3.13 shows the additional type judgments and Table 3.14 shows the additional
typing rules.

\[
\begin{align*}
\Sigma \vdash_k k & \quad \text{Component } k \text{ is well-typed under } \Sigma \\
\Sigma \vdash_t t & \quad \text{Task } t \text{ is well-typed under } \Sigma \\
\Sigma; \kappa \vdash_o o & \quad \text{Object } o \text{ is well-typed under } \Sigma \text{ and } \kappa
\end{align*}
\]

Table 3.13: Type judgments for semantic entities.
A cobox is well-typed if all its elements are well-typed (T-CoBox). Promises are typed similar, but in addition require that the promise identifier is typed to a promise type and that the type of $v_\epsilon$ is a subtype of the parameter type of the promise (T-Promise). Note that configurations are just sets of components, so given a configuration $K$, it can be typed by using the component typing, lifted to sets, i.e., $\Sigma \vdash_k \ast k K$.

Objects are also typed under the reference typing $\Sigma$, but in addition require the identifier $\kappa_b$ of their owning cobox (T-Obj). This identifier is used to construct the reference $\kappa_b.t$ of the object. This reference must be typed under $\Sigma$ to the same class $c$ as given by the object. In addition, the types of the values $\nu$ of the object must be subtypes of the types of the corresponding fields of $c$.

Futures are also typed under the identifier of their owning cobox, but require that the corresponding reference is typed to a future type (T-Fut). In addition, the promise identifier must be typed to a promise type with the same type argument. Finally, the value of the future must be typeable to a subtype of the future’s type argument.

A task is type-correct if its expression $e$ is type correct under an empty variable typing (T-Tsk). This also means that there may be no free variables in $e$. References are simply typed by querying the reference typing $\Sigma$ (T-Ref). The special value $\epsilon$ can be typed to any type which is valid with respect to the underlying (implicit) program (T-Eps).

### 3.4.2 Auxiliary Functions and Predicates

For convenience, we define several functions and predicates on semantic entities.
Definition 3.1 The following predicates distinguish the different objects depending on their kind.

\[
\begin{align*}
\text{transferobj}(o) & \iff \text{transfercl}(c_o) \\
\text{futobj}(o) & \iff o = e\langle\_,\_,\_\rangle \\
\text{normalobj}(o) & \iff \neg (\text{transferobj}(o) \lor \text{futobj}(o))
\end{align*}
\]

It is often required that we state conditions over the set of references, which appear in certain terms.

Definition 3.2 (\textit{refs}) Let \(e\) be an expression. The set of references \(r\), appearing anywhere in \(e\) are denoted by \(\text{refs}(e)\). In addition, we lift \textit{refs} to objects, tasks, components, and configurations as follows. Function \textit{orefs}, is a special variant of \textit{refs}, which only regards objects in a set \(O\) and no futures.

\[
\begin{align*}
\text{orefs}(O) & \triangleq \{ r \mid o\langle\_\_\_\rangle \in O \land r \in \bar{v}\} \\
\text{refs}(O) & \triangleq \text{orefs}(O) \cup \{ r \mid e\langle\_\_\_\rangle \in O \land (r = \kappa_p \lor r = v_e)\} \\
\text{refs}(\tau\langle e\rangle) & \triangleq \text{refs}(e) \\
\text{refs}(b\langle\_\_\_\rangle, O, T, \bar{t}) & \triangleq \text{refs}(O) \cup \bigcup_{t \in UT} \text{refs}(t) \\
\text{refs}(p\langle\_\_\_\rangle, O, v_e) & \triangleq \text{refs}(O) \cup \{ r \mid r = v_e\} \\
\text{refs}(K) & \triangleq \bigcup_{k \in K} \text{refs}(k)
\end{align*}
\]

Definition 3.3 (\textit{rdom}) We define a function \textit{rdom} to obtain the set of the possible references from a configuration. First, we define it on a set of objects, where the object identifiers get prefixed with a certain component identifier.

\[
\text{rdom}(\kappa, O) \triangleq \{ \kappa.t \mid t \in \text{oids}(O)\}
\]

We now define the function \textit{rdom} on components and configurations:

\[
\begin{align*}
\text{rdom}(p\langle\_\_\_\rangle, O) & \triangleq \text{rdom}(\kappa_p, O) \cup \{ \kappa_p \} \\
\text{rdom}(b\langle\_\_\_\rangle, O) & \triangleq \text{rdom}(\kappa_b, O) \\
\text{rdom}(K) & \triangleq \bigcup_{k \in K} \text{rdom}(k)
\end{align*}
\]

By using the \textit{rdom} function we can now define the set of far references that appear in a certain component.

Definition 3.4 (\textit{farrefs}) Let \(k\) be a component. The set of far references that appear anywhere in \(k\) is denoted by \textit{farrefs}(\(k\)) and defined as follows.

\[
\text{farrefs}(k) \triangleq \{ r \mid r \in \text{refs}(k) \land r \notin \text{rdom}(k)\}
\]
Finally, we define function \( \text{obj} \) to find the object in a configuration that is referenced by a certain reference.

**Definition 3.5 (\( \text{obj} \))** Given a reference \( r = \kappa.\iota \), and a configuration \( K \), the function \( \text{obj} \) denotes the object that is referenced by \( r \).

\[
\text{obj}(\kappa.\iota, K) \triangleq o, \text{ if } \text{id}_o = \iota \land \exists k \in K \cdot \text{id}_k = \kappa \land o \in O_k
\]

### 3.4.3 Properties of Data Transfer

When data is transferred between coboxes in JCoBox\(^c\), it is always handled by the \( \text{copy} \) function (cf. Table 3.10), which copies potentially referenced transfer objects and futures. In this section we present several properties of the \( \text{copy} \) function. Many properties depend on the properties of function \( \text{reach} \), which are given in Section A.2.1.

The first property is, that \( \text{copy} \) is total and well-defined. This is not immediately clear from its definition, because \( \text{copy} \) is defined in a recursive way. The property is important as otherwise the operational semantics may become stuck when trying to transfer data.

**Property 3.1** \( \text{copy} \) is total and well-defined.

**Proof.** By showing that these properties hold for the \( \text{reach} \) function (see Property A.1). □

The next property states that all objects that are returned by \( \text{copy} \) are either transfer objects or unresolved futures.

**Property 3.2** Assume \( \text{copy}(\kappa, O, \overline{\nu}, \kappa', O'') = (O', \overline{\nu'}) \). Then for all \( o \in O' \) either \( \text{transferobj}(o) \) or \( \text{futobj}(o) \) \( \land \nu_o = \epsilon \)

**Proof.** Directly by the property that the object set of \( \text{reach} \) only contains transfer objects and unresolved futures (see Property A.2). □

Another property of \( \text{copy} \) is that all object copies have identifiers that are distinct to the target object set.

**Property 3.3** Assume \( \text{copy}(\kappa, O, \overline{\nu}, \kappa', O'') = (O', \overline{\nu'}) \). Then \( \text{oids}(O') \cap \text{oids}(O'') = \emptyset \).

**Proof.** Immediately by the definition of \( \text{copy} \) because all objects of the result set have fresh identifiers. □

The following lemma shows that all references that are newly introduced by \( \text{copy} \) refer to transfer objects or futures that belong to the newly created object set. This means, in particular, that no newly introduced reference can be a far reference.
**Lemma 3.1** Assume \( \text{copy}(\kappa, O, \overline{v}, \kappa', O'') = (O', \overline{v}') \). Then for all \( \kappa'.t \in \text{refs}(O') \cup \overline{v}' \), it holds that either \( \kappa'.t \in \text{refs}(O) \cup \overline{v} \), or \( \kappa'' = \kappa' \) and either \( o(t, c, \_ ) \in O \) and \( o(t, c, \_ ) \in O' \) and \( \text{transfercl}(c) \), or \( f(t, \kappa_p, v_e) \in O \) and \( f(t, \kappa_p, e) \in O' \).

**Proof.** Let \( \kappa'.t \in \text{refs}(O') \cup \overline{v}' \). If \( \kappa'.t \notin \text{refs}(O) \cup \overline{v} \) then nothing has to be shown. So we assume \( \kappa'.t \notin \text{refs}(O) \cup \overline{v} \). But if \( \kappa'.t \) did not already exist, it must have been created by the \( \text{copy} \) function. The \( \text{copy} \) function creates new references by using the substitution \( \sigma \), which is defined as follows. Let \( O'' = \text{reach}(O, \overline{v}) \), then

\[
\sigma = \{ \kappa.t \mapsto \kappa'.t', t \mapsto t' \mid t \in \text{oids}(O'') \wedge t' \text{ fresh} \}
\]

So it only takes existing references and applies the given substitution to it. In particular, it only introduces new references \( \kappa''.t \), where \( \kappa'' = \kappa' \), which is the first condition we had to show. The second condition is given by Property 3.2, which states that all objects in \( O' \) are either transfer objects or unresolved futures.

Finally, we show another main lemma of the \( \text{copy} \) function, namely that there are no references from the original object set into the copied set, where it is assumed that the copied object set belongs to the given target cobox and that references to objects of the target cobox are all referring to previously existing objects of the target cobox.

**Lemma 3.2** Assume \( \text{copy}(\kappa, O, \overline{v}, \kappa', O'') = (O', \overline{v}') \) and \( \forall \kappa'.t \in \text{refs}(O). t \in \text{oids}(O'') \), then \( \neg \exists \kappa'.t \in \text{refs}(O) \) with \( t \in \text{oids}(O') \).

**Proof.** This can be directly concluded from Property 3.3.

### 3.4.4 Well-Formed Configurations

We now give a formal definition of well-formed configurations. Well-formedness can be regarded as a precondition of well-typedness. It states invariants on the operational semantics, which always holds, regardless of the typing of programs.

**Definition 3.6 (Well-Formed Configurations)** A configuration \( K \) is well-formed, denoted by \( \text{wf}(K) \), if it can be derived by rule (WF-CONFIGURATION), given in Table 3.15. Well-formed configurations thus have the following properties:

1. all components have unique identifiers (WF-UNIQUECOMPIDS);
2. all objects of a cobox have component-unique identifiers (WF-UNIQUEOBJIDS), and all suspended tasks exclusively claim a future (WF-SUSPENDSET);
3. unresolved promises have an empty object set (WF-PROMINIT);
4. all objects of a resolved promise are either futures or transfer objects and have component-unique identifiers (WF-PROMRES);
5. far references can never target transfer objects and futures (WF-TRANSFER).
3.4 Type Soundness

We now show that the well-formedness properties are preserved by the operational semantics of JCoBox\(^C\).

**Lemma 3.3 (Well-Formedness Invariant)** Let \( K \) be a configuration with \( \text{wf}(K) \) and let \( K' \) be a configuration with \( K \longrightarrow K' \), then \( \text{wf}(K') \).

**Proof.** The proof is by a case analysis on all reduction rules, where for each rule it is shown that the rule preserves the well-formedness property. The most complexity of the proof is due to the copying of objects between components. See Section A.2.2 for a detailed proof.

### 3.4.5 Preservation Lemma

The first important lemma for showing type soundness is the **Preservation Lemma**, which states that if a well-typed configuration can do a step in the operational semantics of JCoBox\(^C\), the resulting configuration is well-typed again. That is, the operational semantics does not invalidate the well-typedness of configurations.

**Definition 3.7 (Well-Typed Configurations)** A configuration \( K \) is **well-typed** under reference typing \( \Sigma \), written \( \Sigma \vDash K \), if all components of \( K \) are correctly typed under

---

**Table 3.15: Well-Formed Configurations.**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>(WF-CONFIGURATION)</td>
<td>( \forall k \in K \cdot \text{wf}(k) \land \text{wftransfer}(K, k) \land \text{uniqueids}(K) )</td>
</tr>
<tr>
<td>(WF-CoBox)</td>
<td>( \text{uniqueids}(O) \land \text{wfsuspendset}(T) )</td>
</tr>
<tr>
<td>(WF-PROMINIT)</td>
<td>( \text{wf}(\rho(k_p, \emptyset, \epsilon)) )</td>
</tr>
<tr>
<td>(WF-PROMRES)</td>
<td>( \forall o \in O \cdot \text{futobj}(o) \lor \text{transferobj}(o) )</td>
</tr>
<tr>
<td>(WF-TRANSFER)</td>
<td>( \forall r \in \text{farrefs}(k) \cdot \text{normalobj}(\text{obj}(r, K)) )</td>
</tr>
<tr>
<td>(WF-SUSPENDSET)</td>
<td>( \forall t \in T \cdot t = \tau(e'[r.get]) )</td>
</tr>
<tr>
<td>(WF-UNIQUECOMPIDS)</td>
<td>( \forall k, k' \in K \cdot \text{id}<em>k = \text{id}</em>{k'} \Rightarrow k = k' )</td>
</tr>
<tr>
<td>(WF-UNIQUEOBJIDS)</td>
<td>( \forall o, o' \in O \cdot \text{id}<em>o = \text{id}</em>{o'} \Rightarrow o = o' )</td>
</tr>
</tbody>
</table>

69
Chapter 3 Core Calculus

\[ \Sigma, \text{ and the set of references in the domain of } \Sigma \text{ is equal to all legal references of } K. \]

\[ \Sigma \models K \iff \text{wf}(K) \land \text{dom}(\Sigma) = \text{rdom}(K) \land \Sigma \vdash^* K \]

**Definition 3.8 (Reference Typing Extension)** \( \Sigma' \) extends \( \Sigma \), written \( \Sigma \subseteq\Sigma' \) if all references in the domain of \( \Sigma \) are typed to the same type in \( \Sigma' \).

\[ \Sigma \subseteq \Sigma' \iff \forall r \in \text{dom}(\Sigma). \Sigma(r) = \Sigma'(r) \]

Note that this implies \( \text{dom}(\Sigma) \subseteq \text{dom}(\Sigma') \).

**Lemma 3.4 (Preservation)** Let \( p \) be the implicit, fixed program with \( \vdash_p p \). Let \( K_n \) be a configuration and \( \Sigma \) a reference typing with \( \Sigma \models K_n \). Assume \( K_n \rightarrow K_{n+1} \). Then there exists a \( \Sigma' \) with \( \Sigma \subseteq \Sigma' \) and \( \Sigma' \models K_{n+1} \).

**Proof.** By case analysis on the reduction rules (see Proof 32 on Page 170 for a detailed proof).

3.4.6 Progress Lemma

The second important property of JCoBox is the *Progress Lemma*. It states that a well-typed configuration is either a *terminal configuration*, or it can do a step in the operational semantics of JCoBox. Before we define the lemma, we first give a precise definition of a terminal configuration.

**Broken Tasks**

In this subsection we precisely define the configurations in which a task is *broken*. A task is broken if it is in a correctly typed state, but cannot make any progress. Typically, in practice, these situations result in exceptions at runtime. Broken tasks are not prevented by the type system of JCoBox.

**Definition 3.9 (Null Access)** An expression \( e \) is a *null access*, denoted by \( \text{nullacc}(e) \), iff \( \text{null} \) is used as a target.

\[ \text{nullacc}(e) \iff e \in \{ \text{null.f}, \text{null.f} = v, \text{new c in null}, \text{null.m(v)}, \text{null!m(v)}, \text{null.await}, \text{null.get}, \text{null.fut}, \text{null.resolve v} \} \]

**Definition 3.10 (Far Direct Access)** Let \( \kappa_b \) be the identifier of a cobox. An expression \( e \) is a *far direct access*, with respect to \( \kappa_b \), denoted by \( \text{fdaccess}(\kappa_b, e) \), iff the target of the direct access is a far reference.

\[ \text{fdaccess}(\kappa_b, e) \iff e \in \{ \kappa'_b.t.f, \, \kappa'_b.t.f = v, \, \kappa'_b.t.m(v) \} \land \kappa_b \neq \kappa'_b \]
3.4 Type Soundness

Definition 3.11 (Broken Resolve) A resolve expression is broken if an already resolved promise is tried to be resolved again.

\[ \text{brokenresolve}(K, e) \overset{\text{def}}{=} e = \kappa_p \cdot \text{resolve} \nu \land \rho(\kappa_p, O, \nu') \in K \]

Definition 3.12 (Broken Expression) Let \( K \) be a configuration and \( \kappa_b \) the identifier of a cobox. An expression \( e \) is broken, denoted by \( \text{brokenexpr}(K, \kappa_b, e) \), if it is either a null access, a far direct call, or a broken resolve.

\[ \text{brokenexpr}(K, \kappa_b, e) \overset{\text{def}}{=} \text{nullacc}(e) \lor \text{fdaccess}(\kappa_b, e) \lor \text{brokenresolve}(K, e) \]

Definition 3.13 (Broken Task) A task \( t \) is broken if the current expression in the evaluation context of \( t \) is broken.

\[ \text{brokentsk}(K, \kappa_b, \tau(e_{\square}[e])) \overset{\text{def}}{=} \text{brokenexpr}(K, \kappa_b, e) \]

Blocked Tasks

Besides being broken, a task can also be blocked. A blocked task is a task that exclusively waits for an unresolved future and can thus make no progress. In contrast to a broken task, a blocked task is not blocked forever, but can potentially make progress again if the corresponding future gets resolved.

Definition 3.14 (Blocked Task) Let \( b \) be a cobox. A task \( t \) of \( b \) is blocked, denoted by \( \text{blockedtsk}(b, t) \), if \( t \) exclusively waits for an unresolved future.

\[ \text{blockedtsk}(b, t) \overset{\text{def}}{=} t = \tau(e_{\square}[\kappa_b.t.get]) \land r(t, _, e) \in O_b \]

Inactive CoBoxes

Equipped with the definitions of broken and blocked tasks, we can now define inactive coboxes.

Definition 3.15 (Inactive CoBox) Given a configuration \( K \), a cobox \( b \) is inactive, denoted by \( \text{inactive}(K, b) \), iff all its suspended tasks are blocked and \( b \) has either no active task, or the active task is either blocked or broken.

\[ \text{inactive}(K, b) \overset{\text{def}}{=} (\forall t \in T \cdot \text{blockedtsk}(b, t)) \land (\bar{t} \cdot t' \Rightarrow (\text{blockedtsk}(b, t') \lor \text{brokentsk}(K, \kappa_b, t')) \]

where \( b = b(\kappa_b, O, T, \bar{t}) \)
With this definition, we can now state that an inactive cobox cannot do any local steps.

**Lemma 3.5** Let $K$ be a configuration and $b \in K$ be an inactive cobox. Then there is no $b'$ with $b \rightarrow b'$. ■

**Proof.** Let $b = b(\kappa_b, O, T, \bar{t})$ with $b \in K$ and $\text{inactive}(K, b)$. By the definition of inactive, we get

$$\forall t \in T \cdot \text{blockedtsk}(b, t) \quad (1)$$

$$\bar{t} = \bar{t'} \cdot t' \Rightarrow (\text{blockedtsk}(b, t') \lor \text{brokentsk}(K, \kappa_b, t')) \quad (2)$$

By (1), all suspended tasks are blocked and thus are of the form $\tau(e)[\kappa_b, t, \text{get}]$. By (2), either $b$ has no active task, i.e., $\bar{t} = \bullet$, or it has an active task $t'$, i.e., $\bar{t} = \bar{t'} \cdot t'$ and $t'$ is either blocked or broken. In all cases at most two cobox-local rules can apply, namely (R-TASKRESUME) if $T$ is not empty, and (R-FUTGET) if there is an active task $t'$. But both rules require a corresponding resolved future, which is not allowed by the definition of blocked. There is no rule that applies for a broken active task. ■

**Terminal Configurations**

Before we define terminal configurations, we proceed by defining the set of resolvable futures of a configuration.

**Definition 3.16 (Existing Futures)** Let $K$ be a configuration. Then the set of futures existing in $K$ is denoted by $\text{futs}(K)$ and defined as

$$\text{futs}(K) \triangleq \{ o \mid \text{futobj}(o) \land o \in O_b \land b \in K \}$$

**Definition 3.17 (Resolvable Futures)** Given a configuration $K$ the set of resolvable futures of $K$, denoted by $\text{resolvableFuts}(K)$, is the set of unresolved futures whose promises are resolved.

$$\text{resolvableFuts}(K) \triangleq \{ o \in \text{futs}(K) \mid o = f(\iota, \kappa_p, \epsilon) \land p(\kappa_p, \_, \nu) \in K \}$$

Finally, we can define terminal configurations.

**Definition 3.18 (Terminal Configuration)** A configuration $K$ is terminal, denoted by $\text{terminal}(K)$, if all coboxes of that configuration are inactive and there is no resolvable future.

$$\text{terminal}(K) \triangleq \forall b \in K \cdot \text{inactive}(K, b) \land \text{resolvableFuts}(K) = \emptyset$$
Progress Lemma

Having defined terminal configuration, we can now formulate the Progress Lemma, which states that if a well-typed configuration is not terminal, then it can make a step in the operational semantics.

**Lemma 3.6 (Progress)** Let \( p \) be the implicit, fixed program with \( \vdash_p p \). Let \( K_n \) be a configuration which is not terminal, and \( \Sigma \) be a reference typing with \( \Sigma \models K_n \). Then there is a \( K_{n+1} \) with \( K_n \rightarrow K_{n+1} \).

**Proof.** By structural induction on the form of \( K_n \) (see Pages 174ff for a detailed proof).

### 3.4.7 Type Soundness

The Preservation and Progress Lemmas can now be combined to the Type Soundness Theorem, which states that a well-typed configuration is either terminal, or it can do a step in the operational semantics and the resulting configuration is again well-typed.

**Theorem 3.1 (Type Soundness)** Let \( p \) be a well-typed program with \( \vdash_p p \). Let \( K \) be a configuration, and \( \Sigma \) be a reference typing, with \( \Sigma \models K \). Then either \( K \) is terminal, or there is a \( \Sigma' \) with \( \Sigma' \models K' \).

**Proof.** If \( K \) is not terminal, then, by Lemma 3.6, there exists a \( K' \) with \( K \rightarrow K' \). By Lemma 3.4, there exists a \( \Sigma' \) with \( \Sigma' \models K' \).

### Lemma 3.7 (Initial CoBox Typing)

Let \( p \) be a well-typed program with \( \vdash_p p \), and let \( b^\text{init}_p \) be the initial cobox of \( p \), then \( \emptyset \models b^\text{init}_p \).

**Proof.** Let \( p = D e \). The initial cobox of \( p \) is defined as \( b(\text{main}, \emptyset, \emptyset, \tau(e)) \). From \( \vdash_p p \) we obtain \( \emptyset \vdash_e e : \tau \) by (T-PROGRAM). By (T-TSKE), \( \emptyset \vdash_t \tau(e) \). As also \( \emptyset \vdash^* \emptyset \) and \( \emptyset \vdash^* \emptyset \) we get, by (T-COBOX), \( \emptyset \vdash_k b(\text{main}, \emptyset, \emptyset, \tau(e)) \). As \( \text{refs}(b^\text{init}_p) = \emptyset \) we finally conclude \( \emptyset \models b^\text{init}_p \).

### Corollary 3.1

Let \( p \) be a well-typed program with \( \vdash_p p \). Let \( b^\text{init}_p \) be the initial cobox of \( p \), and \( K \) a configuration with \( b^\text{init}_p \rightarrow^* K \). Then there exists a \( \Sigma \) with \( \Sigma \models K \) and either \( K \) is terminal, or there is a \( K' \) with \( K \rightarrow K' \).

**Proof.** By induction on the number of reduction steps. The base case results from Lemma 3.7, the induction step is by Theorem 3.1.

### 3.5 Properties

This section shows further properties of JCoBox\(^c\).
3.5.1 Data Races

A data race is a situation where a single memory location is accessed simultaneously by two threads and at least one access is a write [SBN+97]. As the operational semantics of JCoBox is an interleaving semantics, there are never true simultaneous accesses in JCoBox. However, we can define what a simultaneous access in the context of JCoBox means, namely, that there are at least two active tasks in a configuration which have a field access as the current evaluated expression and both tasks can proceed. We first define what it means to have a field access in a cobox that can proceed.

Definition 3.19 (Field Access) Let $K = K' \cup b$. Configuration $K$ has a field access in cobox $b$ denoted by $\text{fieldaccess}(K, b)$, iff $b = \text{b}(\kappa_b, O, T, \bar{\tau} \cdot \tau\langle e[r.f]\rangle)$, with $e \equiv r.f \lor e \equiv r.f = v$, and $K \rightarrow K''$, with $K'' = K''' \cup b'$ and $b' = \text{b}(\kappa_b, O, T, \bar{\tau} \cdot \tau\langle e[v]\rangle)$.

Definition 3.20 (Simultaneous Field Access) Let $K$ be a configuration. $K$ has a simultaneous field access iff $K = K' \cup b \cup b'$ with $\text{fieldaccess}(K, b)$ and $\text{fieldaccess}(K, b')$.

We now show that in JCoBox, there can never be simultaneous field accesses in a well-formed configuration, by showing that if there is can be at most one task with the same field access that can proceed.

Lemma 3.8 (There are no Simultaneous Field Accesses) Let $K$ be a well-formed configuration. Then there cannot be any simultaneous field access in $K$.

Proof. We show that if there are more than one cobox with field accesses to the same field, than at most one of these coboxes can make a step. Let $K$ be a well-formed configuration. Let $r$ be the object which is accessed simultaneously. Let $K' \subseteq K$ be the set of coboxes which have an active task of the form $\tau\langle e[r.f]\rangle$. Without loss of generality, we assume that all simultaneous field accesses are read accesses. Then at most one of these read access can be reduced to a value $v$, all others are broken.

By case distinction on the form of $r$.

Case $r = \text{null}$. In that case no task can proceed as $(\text{R-FieldSelect})$ requires a non-null reference.

Case $r = \kappa_b.\iota$. In that case, $(\text{R-FieldSelect})$ requires that the cobox of the task has identifier $\kappa_b$. By the well-formedness condition, there can be at most one cobox with identifier $\kappa_b$. Thus, there can be at most one active task which has form $\tau\langle e[r.f]\rangle$, which is not broken. All other active tasks, which have the form $\tau\langle e[r.f]\rangle$ must belong to different coboxes and thus are broken.

Corollary 3.2 (Data Race Freedom) Data races are not possible in JCoBox.

Proof. Directly by Lemma 3.8 as a data race requires a simultaneous field access.
3.5 Properties

3.5.2 Determinism

\(\alpha\)-Equivalence

Before we show the actual determinism results, we first define the standard \(\alpha\)-equivalence of configurations. \(\alpha\)-equivalence is needed, because identifiers of objects and components are chosen by the rules in a non-deterministic way. With \(\alpha\)-equivalence two terms are equal, if they are equal modulo renaming of identifiers. We first define a renaming function \(\alpha\).

**Definition 3.21 (Renaming Function \(\alpha\))** A renaming function \(\alpha\) is an injective function that maps global identifiers to global identifiers and object references to object references and has the following property:

\[
\forall \kappa, \iota \in \text{dom}(\alpha) \cdot \alpha(\kappa, \iota) = \alpha(\kappa, \iota'), \text{ for some } \iota'
\]

In other words, the mapping of object references must be consistent with the mapping of global identifiers. Given a renaming function \(\alpha\) we write \(\alpha(\kappa, \iota) \equiv \iota'\) if \(\alpha(\kappa, \iota) = \kappa' \cdot \iota'\).

**Definition 3.22 (\(\alpha\)-Renaming)** Let \(\alpha\) be a renaming function. The function \(rn_\alpha\) defines how \(\alpha\) is applied to semantic terms, where the application of \(rn_\alpha\) to an expression \(e\) is equivalent to a substitution of all occurrences of references in \(e\) by the mapped references defined by \(\alpha\).

\[
\begin{align*}
    rn_\alpha(K \cup k) &\equiv rn_\alpha(K) \cup rn_\alpha(k) \\
    rn_\alpha(\nu(\kappa_p, O, v_\epsilon)) &\equiv \nu(\alpha(\kappa_p), rn_\alpha(\kappa_p, O), rn_\alpha(v_\epsilon)) \\
    rn_\alpha(\omega(\kappa_b, O, T, \bar{T})) &\equiv \omega(\alpha(\kappa_b), rn_\alpha(\kappa_b, O), rn_\alpha(T), rn_\alpha(\bar{T})) \\
    rn_\alpha(T \cup t) &\equiv rn_\alpha(T) \cup rn_\alpha(t) \\
    rn_\alpha(\bar{T} \cdot t) &\equiv rn_\alpha(\bar{T}) \cdot rn_\alpha(t) \\
    rn_\alpha(\tau(e)) &\equiv \tau(rn_\alpha(e)) \\
    rn_\alpha(\kappa, O \cup o) &\equiv rn_\alpha(\kappa, O) \cup rn_\alpha(\kappa, o) \\
    rn_\alpha(\kappa, o(\iota, c, \bar{v})) &\equiv o(\alpha(\kappa, \iota), c, rn_\alpha(\bar{v})) \\
    rn_\alpha(\kappa, \nu(\iota, \kappa_p, v_\epsilon)) &\equiv \nu(\alpha(\kappa, \iota), \alpha(\kappa_p), rn_\alpha(v_\epsilon))
\end{align*}
\]

Equipped with the renaming function and its application, we can now define \(\alpha\)-equivalence.

**Definition 3.23 (\(\alpha\)-Equivalence)** Two configurations \(K, K'\) are \(\alpha\)-equivalent, denoted by \(K \equiv_\alpha K'\), if there exists a renaming function \(\alpha\) with \(K = rn_\alpha(K')\). In the analogous way \(\alpha\)-equivalence is defined for components \(k\).
Sequential Configurations

We now show that the execution of configurations that only consist of a single cobox is deterministic. Deterministic means that all configurations that can be derived in the next step are \(\alpha\)-equivalent.

We first show that the cobox-local rules are deterministic, when a cobox has no futures.

**Lemma 3.9 (CoBox-Local Determinism Modulo Futures)** Let \(b\) be some cobox, which has no futures. Then for all \(b', b''\) with \(b \rightarrow_b b'\) and \(b \rightarrow_b b''\), it holds that \(b' \equiv_\alpha b''\).

**Proof.** As \(b\) has no futures, rules \((\text{R-TASKRESUME})\) and \((\text{R-FUTGET})\) cannot be applied. All remaining rules have disjunct preconditions, because they all require an active task, but with disjunct forms. This means that at most one rule can apply. We now have to show that each rule is deterministic. Except rule \((\text{R-NEWOBJLOCAL})\) all rules reduce to a unique next cobox, because no non-determinism is involved. Rule \((\text{R-NEWOBJLOCAL})\) non-deterministically chooses a new object identifier for the newly created object. Let \(b'\) and \(b''\) be two coboxes derived by the rule \((\text{R-NEWOBJLOCAL})\). Let \(\iota'\) be the object identifier chosen for \(b'\) and \(\iota''\) be the object identifier chosen for \(b''\). Let \(\kappa_b = \text{id}_b\), then we define the renaming function \(\alpha\) as the identity function, except that \(\alpha(\kappa_b \cdot \iota'') = \kappa_b \cdot \iota'\). We can then obtain \(b' = \text{rn}_\alpha(b'')\), hence \(b' \equiv_\alpha b''\). \(\Box\)

**Lemma 3.10 (Single-CoBox Determinism)** Let \(K\) be a well-formed configuration with \(K = \{ b \}\). Then for all \(K', K''\) with \(K \rightarrow K'\) and \(K \rightarrow K''\), it holds that \(K' \equiv_\alpha K''\).

**Proof.** By the assumption that \(K\) only consists of a single cobox, only rules can be applied that do not require additional components. As we require that \(K\) is well-formed, i.e., \(\text{wf}(K)\), \(b\) cannot have any futures. This is due to the fact that every future refers to some promise \(\rho\). By the well-formedness condition, such a promise must exist in \(K\), which is, however, not possible as \(K\) only consists of a single cobox. Hence Lemma 3.9 can be applied for all cobox-local rules. We now look at the global rules (cf. Table 3.12). The following rules require no additional components beside a single cobox: \((\text{R-NEWCOBOX})\), \((\text{R-PROMNEW})\), and \((\text{R-ASYNCCALLLOCAL})\). In all these rules the only non-determinism comes from choosing fresh identifiers. Thus, analogously to the proof of Lemma 3.9 we can define for each case an \(\alpha\)-renaming function. \(\Box\)

Lemma 3.10 shows that the execution of a configuration with a single cobox is always deterministic. However, such a configuration can create coboxes and promises, in which case Lemma 3.10 cannot be applied anymore. The creation of coboxes and promises, however, can be prevented by forbidding the use of certain JCoBox features.
Definition 3.24 (Sequential Program) A program \( p = D e \) is sequential iff no class in \( D \) is a cobox class, and \( p \) does neither contain asynchronous method calls nor promise- or future-related expressions.

Note that a sequential program can have yield expressions. However, as there are no asynchronous method calls, there can at most be one task, which means that this task stays active when executing yield.

Lemma 3.11 (Sequential Program Determinism) Let \( p \) be a sequential program with \( \vdash_p p \), and let \( K \) be a configuration with \( b^{p}_{\text{init}} \xrightarrow{*} K \), then for all \( K', K'' \) with \( K \xrightarrow{} K' \) and \( K \xrightarrow{} K'' \) it holds that \( K' \equiv_a K'' \).

Proof. This is obtained by Lemma 3.10, the fact that \( b^{p}_{\text{init}} \) is a configuration with only a single cobox, and the fact that new components cannot be created by the given assumptions and thus all derived configurations only consist of a single cobox again.

3.5.3 Sequential Java Equivalence

We now show that if we restrict the possible JCoBox\(^c\) programs to Java-like programs, then these programs will behave in JCoBox\(^c\) like in a standard Java semantics.

Definition 3.25 (Java-like Program) A program \( p \) is Java-like if it only uses standard Java expressions, and all classes are plain.

Sequential Java Semantics

Table 3.16 gives the definition of a standard Java operational semantics, denoted by \( \xrightarrow{\rightarrow}_J \). It is defined on pairs \( O, e \) consisting of an object heap \( O \) and an expression \( e \) to be reduced. The semantics uses references of the form \( \text{MAIN.}ι \), even though there are no coboxes and thus it would be enough to simply use object identifiers as references. However, to be compatible to the JCoBox\(^c\) semantics, references are prefixed with a cobox identifier, which, however, has no influence on the semantics. By inspecting the rules it should be clear that this semantics corresponds to standard Java-core semantics like ClassicJava [FKF99], for example. In the following we call this Java-like language \( \text{JavaSeq} \).

Initial Configuration. Given a Java-like program \( p = D e \) the initial JavaSeq configuration is simply defined as \( (\emptyset, e) \).

Sequential Java Equivalence

To be able to relate the different operational semantics, we define an equality on the different configurations, which is based on alpha-renaming.
Then either Corollary 3.3

Let $\alpha$ be a reference typing. Let $\Sigma \vdash_\alpha O$ and $\Sigma \vdash_\alpha e$, and $b = b(\kappa_b, O', \emptyset, \tau(e'))$ some cobox with $\Sigma \vdash b$ and $\langle O, e \rangle \equiv_\alpha^J b$. Then either

1. $e = \nu$ and $e' = \nu'$, or
2. $e = e_\square[e''']$, $e' = e_\square[e'''']$, $\text{nullacc}(e'''')$, and $\text{nullacc}(e''')$, or
3. there exists a $\langle O'', e''' \rangle$ with $\langle O, e \rangle \rightarrow_J \langle O'', e''' \rangle$, and there exists a $b'$ with $b \rightarrow b'$, and $\langle O'', e''' \rangle \equiv_{\alpha}^J b'$.

Proof. The proof is by structural induction on the form of $e$. Without loss of generality, we assume that $\langle O, e \rangle \equiv_\alpha^J b$ where $\alpha$ is the identity function. Note that then $e = e'$ and $O = O'$. It is then straightforward to show the lemma.

Corollary 3.3 Let $p = D e$ be a type-correct Java-like program. Let $\langle \emptyset, e \rangle$ be the initial JavaSeq configuration of $p$ and let $b_{\text{init}}^p$ be the initial JCoBox$^c$ configuration of $p$. Then for all $\langle O, e \rangle$ with $\langle \emptyset, e \rangle \rightarrow_J^* \langle O, e \rangle$ there exists a $b'$ with $b_{\text{init}}^p \rightarrow^* b'$ and $\langle O, e \rangle \equiv_\alpha^J b'$.
Proof. The proof is by induction on the length of the derivation, where the base case is proved by showing that initial configurations are $\alpha$-equivalent and the induction step is shown by using Lemma 3.12.

3.6 Discussion and Related Work

The formalization of JCoBox$^C$ is based on several other existing formalizations. The sequential part of the dynamic semantics is based on Featherweight Java [IPW01] and ClassicJava [FKF99]. The combination of futures and cooperative multitasking is similar to that of Creol [dBBCJ07]. The treatment of transfer objects is similar to that of passive objects in ASP [CHS04]. The decoupling of asynchronous method calls, promises, and futures is similar to Ábrahám et al. [ÅGGS09].

The formalization is also similar to a previous calculi for the cobox model [SPH08], which treats hierarchical coboxes. It is based on a labeled transition semantics with explicitly modeled messages. This allows for describing the behavior of composite coboxes. In addition, parent coboxes can block external communication to nested coboxes. The hierarchical treatment significantly complicates the operational semantics. The additional blocking feature was disregarded in this thesis, as our experiences showed that the blocking feature is rarely needed in practice and greatly complicates the implementation. The formalization in this thesis also integrates promises and transfer objects—features not present in [SPH08]. Lifting the semantics of JCoBox$^C$ to hierarchical coboxes is left as future work.
CHAPTER 4

The JCoBox Language

One of the goals of this thesis is to develop a programming model that can be efficiently implemented and that can be used in practice. Java [GJSB05] is an object-oriented programming language, which is widely used in practice. It is statically typed and has a platform-independent concurrency and memory model [MPA05]. We thus decided to realize the cobox model on top of Java and call it JCoBox. The core of JCoBox is similar to the calculus presented in Chapter 3. It only has some syntactical differences, and also differs in the treatment of synchronous calls and field accesses, which are explained in this chapter. In addition to the core features, JCoBox also supports Java features like generics, arrays, static members, and exceptions, for example. JCoBox is a conservative extension of sequential Java, which means that a well-formed sequential Java program is a well-formed JCoBox program. There is only one exception, which is that static fields in JCoBox have some restrictions (see Section 4.2.3). The syntax of Java is only minimally extended by an additional operator for asynchronous method calls and an extended new expression. All other extensions are done by standard Java annotations or use special purpose classes and interfaces. We assume that the reader is familiar with sequential Java and thus only explain the concepts introduced by the JCoBox extension.

4.1 Core Constructs

This section explains the constructs of JCoBox that are necessary for realizing the cobox model. In particular it describes how

- coboxes are created and objects are assigned to coboxes;
- the different object kinds (standard, transfer, immutable) are distinguished;
- asynchronous method calls, futures, and promises are addressed;
- cooperative multi-tasking is realized.
4.1.1 Object and CoBox Creation

Object Creation

Objects are created in JCoBox by using the `new` expression. By default, an object is created in the current cobox, i.e., the cobox of the executing task. It is possible to create objects in different coboxes by using the extended `new` expression:

```
new <classname>(<arguments>) in (<expr>)
```

In that case, the new object is created in the cobox where the object is located that is referenced by the evaluated value of `<expr>`.

CoBox Creation

CoBoxes are not first-class citizens of the language. So there is no mechanism to explicitly create a cobox or to directly interact with a cobox. CoBoxes are created implicitly when a cobox class is instantiated, using the standard `new` expression. Implicitly means that the actual result of such a creation is a reference to a standard object instance of the cobox class. That object, however, belongs to a new, implicitly created, cobox. This first object is the initial and first service object of the cobox. It can then be used to further interact (indirectly) with the cobox. It is important to note that instances of cobox classes are standard objects in the cobox model. They have no special role compared to other standard objects in the same cobox. A class is a cobox class if it either inherits from another cobox class, or it is annotated with the `@CoBox` annotation. A cobox class has the implicit invariant that only one instance of that class can be in the same cobox. In addition, it is guaranteed that it is the first object of the cobox and that the constructor code is executed by the first task of the cobox. The following JCoBox code defines class `ChatServer` as a cobox class.

```
@CoBox class ChatServer {
  ...
}
```

Note that by using the extended `new` expression, it is also possible to create objects of standard classes in a fresh cobox. JCoBox provides a predefined cobox class `EmptyCoBox`, which can be used for that purpose:

```
new C() in (new EmptyCoBox());
```

Hence, one can see the `@CoBox` annotation as a guarantee that objects of these classes are always created in their own cobox.
4.1.2 Asynchronous Method Calls

Asynchronous method calls are expressed by using the `!` operator instead of a dot. For example, `a ! m()` asynchronously invokes the method `m()` on object `a`. Any method can be invoked asynchronously in JCoBox.

For asynchronous method calls, it is not possible to omit the implicit `this` receiver, i.e. writing `! m()` is not possible. Instead, one has to explicitly add `this` as receiver: `this ! m()`. Otherwise the syntax would be ambiguous as `!` is also used as a boolean negation operator in Java.

Asynchronous method calls indicated by `!` are partially ordered as described in Section 2.2.3. As in a distributed setting (cf. Section 4.2.6), guaranteeing the ordering of messages can impose additional costs, JCoBox also supports an unordered variant of an asynchronous method call, indicated by the `!!` operator. Unordered method calls are executed without any ordering guarantees, in particular, the execution of unordered method calls is not related to ordered once and vice versa. This gives the messaging infrastructure more freedom on how to deliver such messages.

4.1.3 Futures

Asynchronous method calls return futures. Futures in JCoBox are instances of the special interface `Fut<V>` (see Listing C.2, Page 189), where `V` is the type of the future value and corresponds to the return type of the called method. As type arguments in Java can only be reference types, primitive types and `void` cannot be directly handled. In these cases the corresponding wrapper classes are used instead. For example, if the called method has `void` as return type, the class `java.lang.Void` is used as the type argument for the `Fut` interface. The following code invokes an asynchronous method and stores the result in a future variable:

```java
Fut<Session> fut = server ! connect(this);
```

The `Fut` interface offers two methods for claiming the future value as described in Section 2.3.1. The method `get()` claims the future value exclusively, the method `await()` claims the future value cooperatively. The following code claims a future cooperatively by using `await()`:

```java
Session session = fut.await();
```

4.1.4 Synchronous Communication

Synchronous communication can be simulated by asynchronous method calls with an immediate `get()` or `await()`. Contrary to JCoBox, it is also possible in JCoBox to synchronously invoke a method on a far reference. A call `x.m()` is treated in JCoBox like an asynchronous call with an immediate `get()`. That is, `x.m()` is equivalent to
\( x!m().\texttt{get}() \) if \( x \) refers to a far object. Note that the behavior of near synchronous calls correspond to the standard Java stack-based execution as formalized in JCoBox\textsuperscript{C}.

### 4.1.5 Promises

Promises are represented by the interface `Promise\langle V\rangle` (Listing C.4, Page 190), where \( V \) is the type of the promise value. Promises can be created by the static `newPromise()` method. Futures are obtained from a promise by the `getFuture()` method. Resolving the promise is done by invoking the `resolve(V)` method with the value that should be assigned to the future. Resolving a promise can only be done once, all subsequent calls are ignored.

To show how promises can be used, we give an example, which is taken from [HBS73] and translated to JCoBox (cf. Listing 4.1). It implements a simple dating service that brings two lonely hearts together. If the dating service does not immediately find a matching partner, it returns a future to a promise instead (Line 29). The promise is first created by using the `newPromise()` method (Line 27) and then stored in an internal partner data structure. The promise is resolved when a matching partner is found eventually (Line 21). If a partner is immediately found, it is directly returned. Due to strong typing, the result value is wrapped into a future by using the `toFut(V)` method (Line 22).

### 4.1.6 Cooperative Multi-Tasking

Beside suspending a task to wait for a future, a task can also give up control by yielding. Yielding is done in JCoBox by using the `yield()` method. It is typically used for long running tasks to prevent them from completely blocking the cobox for other tasks. For example, a cobox to download a file from the Internet might do this by a task, which reads chunks in a loop:

```java
byte[] chunk = new byte[4096];
while (!stopped) {
    int nbytes = netStream.read(chunk);
    if (nbytes == -1)
        break;
    fileStream.write(chunk,0,nbytes);
    yield();
}
```

After the handling of a single chunk, other tasks can become active in the cobox to stop the downloading process, for example.
4.1 Core Constructs

```java
@CoBox class DatingService {
    static class PartnerRequest {
        Partner partner;
        Attributes attributes;
        Promise<Partner> partnerPromise;
    }

    List<PartnerRequest> partnerdb = new ArrayList<PartnerRequest>();

    Fut<Partner> findPartner(Partner lonelySoul, Attributes attr) {
        PartnerRequest bestPartner = null;
        int bestScore = 0;
        for (PartnerRequest r : partnerdb) {
            int score = r.attributes.matchScore(attr);
            if (score >= bestScore) {
                bestPartner = r; bestScore = score;
            }
        }

        if (bestPartner != null) {
            partnerdb.remove(bestPartner);
            bestPartner.partnerPromise.resolve(lonelySoul);
            return toFut(bestPartner.partner);
        } else {
            PartnerRequest request = new PartnerRequest();
            request.attributes = attr;
            request.partner = lonelySoul;
            request.partnerPromise = newPromise();
            partnerdb.add(request);
            return request.partnerPromise.getFuture();
        }
    }
}

@Immutable class Attributes {
    ... // some fields
    int matchScore(Attributes compare) {
        int score = ... // calculate score
        return score;
    }
}
```

4.1.7 Transfer Classes

To realize transfer objects, classes can be declared as transfer classes. A class is a transfer class if it is either annotated with the `@Transfer` annotation or inherits from another transfer class. Transfer classes can only inherit from `Object` or from other transfer classes. Transfer objects are always copied when they are passed to another cobox. As described by the formal semantics (cf. Section 3.3.3), transfer objects are copied transitively, i.e., the transfer object and all referenced transfer objects are copied.

4.1.8 Immutable Classes

To realize immutable objects, classes can be declared to be immutable. A class is immutable if it is annotated by `@Immutable` or inherits from another immutable class. Immutable classes are only allowed to inherit from `Object` or from another immutable class. All fields of an immutable class are implicitly `final`. This ensures that objects of immutable classes are shallowly immutable. In addition, it must be prevented that immutable objects can refer to transfer objects. JCoBox ensures this by forbidding that the type of the right hand side of assignments to fields of immutable classes are of transfer classes. In addition, futures as types are only allowed if their type argument is not a transfer class. If it statically not clear whether the right hand side is of a transfer class or not, e.g., when it is of type `Object`, it is checked at runtime and an `ImmutabilityException` is thrown in the illegal case. Immutable objects can be shared arbitrarily between coboxes, and, in particular, can be accessed concurrently. Whenever possible, immutable objects should be preferred over transfer objects, as they have no copying overhead when passed between coboxes. It is possible to invoke methods asynchronously on immutable objects. In that case a task is created in the current cobox, which executes the call. As the immutability definition of JCoBox is rather restrictive, it is possible to disable the immutability checks, by using the `@Unchecked` annotation. In this case, the programmer has to manually ensure immutability of the class.

Escaping of this. It is possible in JCoBox to see two different values for a final field because JCoBox does not prevent that a reference to an immutable object escapes during its construction. This can happen, for example, when passing `this` to another object in the constructor. It is then possible to read the default value of a final field before it gets its actual value. We do not see this as a big issue as this is even a problem in standard sequential Java. It is possible to statically prevent this problem, for example, by enforcing that constructors of immutable classes are anonymous [VB01].
4.1.9 Start-Up and Termination of JCoBox Programs

JCoBox programs are started like Java programs by executing the standard main method. The main method is executed by an initial task that runs in a special main cobox, which is created at start up. By default, a JCoBox program terminates when all tasks of all coboxes have terminated. This behavior can be configured by the setAutomaticShutdown(boolean) method. If automatic shutdown is turned off, a program has to be explicitly shutdown by calling the shutdown() method or using the exit(int) method, which has an additional exit code parameter. Disabling the automatic shutdown is often needed when using legacy Java code because threads might be created by legacy code, which are not known to the JCoBox runtime.

4.2 Extensions

One of the goals of this thesis is to develop a practical concurrency model. The JCoBox language introduced in Section 4.1 is already sufficient to write many small applications. However, to be conveniently usable in practice, several additional issues have to be addressed, which are covered by this section.

4.2.1 Timeouts

In practice, it is often required to do something for a certain amount of time. Several operations in JCoBox have optional time parameters, which are described in the following.

Future Operations

Both get() and await() also exist as versions that take timeout arguments, throwing a TimeOutException when a timeout happens while waiting for a future. This is especially useful in the distributed setting, for example:

```java
try {
    Session session = fut.await(5, TimeUnit.SECONDS);
} catch (TimeOutException e) {
    System.out.println("Connection to server timed out");
}
```

Yielding and Sleeping

The yield() method can be called with optional time parameters to specify a period of time to wait before the yielded task is added to the ready queue again. For example, to constantly send keepAlive messages to a server session, a task can yield for one second before sending the next keepAlive message:
void ensureAliveness() {
    while (!stopped) {
        session!keepAlive();
        yield(1, TimeUnit.SECONDS);
    }
}

The corresponding sleep(int, TimeUnit) method can be used to block the running task for a certain amount of time, which prevents other tasks from becoming active in the meantime.

### 4.2.2 Futures

This subsection presents several features of JCoBox that are related to futures.

**Future Handlers**

The value of futures can be claimed by using either get() or await(). Another possibility is to use event handlers, which are executed when the future is ready. This is similar to when-catch expressions in E [MTS05] and AmbientTalk [VCMB+07].

```java
Fut<Room> roomFut = server!joinRoom("Public");
roomFut.when(new Fut.VoidHandler<Room>() {
    public void ready(Room room) {
        room!publish("Hi!");
    }
    public void error(Thrower e) {
        e.printStackTrace();
    }
});
```

Future handlers can be passed to the `when(Fut.Handler<V,R>)` method of futures. A handler has two type parameters, V for representing the type of the future value and R for representing the return type of the future handler methods. A future handler has two methods, `ready(V)` and `error(Thrower)`. The `ready` method is invoked when the future has been resolved to a value and the `error` method is invoked when the future has been resolved by an exception. In the example, a VoidHandler is used, which just means that the return type of the handler methods is `void`.

The handler code of the above example is semantically equivalent to the following code, which does not use handlers, but instead uses an asynchronous method call and the `await()` method to wait for the future:

```java
Fut<Room> roomFut = server!joinRoom("Public");
this!handle(roomFut);
```

where handle is defined as
void handle(Fut<Room> roomFut) {
    try {
        Room room = roomFut.await();
        room.publish("Hi!");
    } catch (Throwable e) {
        e.printStackTrace();
    }
}

Implementation Remark. In the current implementation of JCoBox, the variant that uses the future handler is more efficient than the variant that uses await(), as await() could require the suspension of the underlying JVM thread.

Future Chaining

In the previous example, the value of the future is not directly needed, but only used as a target for a further asynchronous method call. In that case, it is possible to directly issue the call on the future. These calls are then executed as soon as the future is resolved. We call this mechanism future chaining, but it is better known as promise pipelining [LS88, MTS05]. The example above can be rewritten as:

Fut<Room> roomFut = server.joinRoom("Public");
roomFut.publish("Hi!");

Note that in that case exceptions are propagated to the chained futures. So if the joinRoom call throws an exception, the roomFut future would rethrow that exception if it is claimed.

Future chaining enables greater use of parallelism compared to explicit waiting for the results. Consider the following code that explicitly waits for the future:

Fut<Room> roomFut = server.joinRoom("Public");
roomFut.await() publish("Hi!");

In this version the task has to be suspended until the future is resolved. In the version with future chaining, the task can continue with its execution. The chained call can even be executed in parallel with the task execution. In the distributed setting, future chaining even allows for executing subsequent calls directly on the receiver side, which can drastically improve the performance of such calls [MTS05]. Future chaining in the distributed setting, however, is currently not implemented in JCoBox and left for future work.
4.2.3 Additional Java Features

To be usable in practice, JCoBox has to support the full Java language. This subsection explains how JCoBox integrates extended Java features and what has to be kept in mind when using them.

Interfaces

Interfaces have been used in Chapter 3, but it has not been explained, how they are treated in JCoBox. Like classes, interfaces can be annotated with @CoBox, @Transfer, and @Immutable. Annotated interfaces restrict the possible kinds of implementing classes as these classes are automatically of the kind of their implemented interfaces. It is clear that a class may not implement two interfaces with conflicting annotations. Now consider the interface Comparable. Which class kind should it have? Should classes that implement that interface be standard classes, transfer class, or immutable classes? In fact, one does not want to fix the kind in this case, because it may be usable for every class. Interfaces that do not have any annotation in JCoBox are called unspecific interfaces. Unspecific interfaces leave the concrete kind open, i.e., any class can implement unspecific interfaces. In fact, as all classes always inherit from Object, Object can also be seen to be of an unspecific kind. This now raises the question of how to specify standard interfaces, i.e., interfaces which enforce standard classes as implementers. For this, JCoBox introduces a new annotation @CoObject. In fact, @CoObject can be regarded as the default annotation for classes without any specific kind.

Constructors

In the current version of JCoBox constructors of objects are always invoked synchronously; asynchronous creation of objects is not possible. This is in particular important for constructors of cobox classes as these constructors are executed in a newly created cobox. While the constructor of such a far object is executed, the creating cobox is blocked. It is thus not possible inside a cobox class constructor to synchronously call-back to the creating cobox as this will immediately lead to a deadlock. Asynchronous creation of objects is a useful extension of JCoBox, which is left as future work.

To circumvent the constructor problem, one can create a separate initialization method, which is invoked asynchronously after the object has been created. The asynchronous call can also be directly done inside the class constructor, for example:

```java
@CoBox class C {
    C() { this . init(); }
    void init() { ... }
}
```
It is guaranteed in that case that the \texttt{init} call will be the first executed call after the constructor has finished. As the \texttt{init} method then runs independently from the creating cobox, it does not suffer from the above mentioned problems. Note that this pattern can also be used to implement autonomous objects, which become active when they are created.

**Fields**

In contrast to the core calculus, fields of far objects can be accessed in JCoBox. Field accesses are always synchronous, i.e., it is not possible to asynchronously access a field. The semantics of a far field access can be compared to a synchronous method call of a wrapper method that does the field access. Far field accesses are thus always mutual exclusive and do never lead to data races. However, in general one should avoid to access fields of far objects. Instead, fields should only be accessed via corresponding getter and setter methods, which also enables asynchronous access.

*Final Fields.* Fields that are explicitly declared as \texttt{final} are accessed like fields of immutable objects if it is statically clear that they cannot refer to a transfer object or a future of a transfer object. These fields can be accessed concurrently, without having to wait for the owning cobox to become free.

**Static Fields**

Static fields in JCoBox are treated like fields of immutable classes and have the same restrictions. This means that static fields are implicitly \texttt{final} and the type of the right-hand side of assignments may not be of transfer classes, futures of transfer classes, unspecific interfaces, and \texttt{Object}. This guarantees that global state is immutable in JCoBox and that objects referenced by static fields are guaranteed not to be transfer objects. Treating global state as immutable is also done in the FAN language \cite{Fan09} and Kilim \cite{SM08}. If mutable global state is needed, it can be put into a separate cobox, which is referenced by a \texttt{final static} field. Listing 4.2 illustrates this by means of an example.

**Static Methods**

Static methods can only be called synchronously and are executed in the context of the calling cobox. Static methods can be executed in parallel by tasks of different coboxes. This is safe as global state is always immutable. Executing static methods thus imposes no runtime overhead in JCoBox.

Code of static initializers is executed in the context of the \texttt{main} cobox. It should be avoided to create standard objects in static initializers as otherwise these objects all belong to the \texttt{main} cobox, which might result in unexpected runtime behavior. JCoBox checks this condition for code that is directly included in static initializers.
Chapter 4 The JCoBox Language

```java
class Statics {
    static int x = 5; // OK: primitive type
    static Object o = new GlobalState(); // OK: cobox class
    static Object c = new SomeTransferClass(); // ERROR: transfer class
}

@CoBox class GlobalState {
    static GlobalState instance = new GlobalState();
    public int x; // some global variable accessible by GlobalState.instance.x
}
```

Listing 4.2: Treatment of static fields in JCoBox.

Note that the creation of objects in a static context cannot lead to data races, but might lead to deadlocks because of unexpected cobox structures.

Arrays

Arrays are treated like transfer objects and are copied when passed to another cobox. There are several reasons for this decision. First, it is unlikely that arrays are used as service objects. Second, if arrays would be allowed as service objects, every array access would have to be protected against concurrent access, making array accesses inefficient. But in most cases, arrays are used for performance reasons. When arrays are treated as transfer objects, there is no overhead when accessing arrays as it is guaranteed that the array is local to the current cobox. Finally, Java RMI \[Ora10a\] treats arrays as serializable, which is semantically similar to transfer objects.

If arrays should act as service objects, they can be wrapped into a MutableArray object from the JCoBox library, which can then be passed by reference to other coboxes. In addition, the JCoBox library offers an ImmutableArray class to create immutable array-like objects that can be safely accessed by multiple coboxes.

Implementation Remarks. There is another, implementation-related aspect, why arrays are treated as transfer objects. The reason is that standard objects require an additional field to refer to their owning cobox. As arrays cannot inherit from other classes then Object, it is difficult to add such a field to arrays. The alternative, to use a global map that manages the array-cobox relation, does not scale in a concurrent setting.

Exceptions

Exceptions are supported by JCoBox like in Java. However, the question is how and when to handle exceptions thrown in asynchronous method calls.
The first way is to claim the future of an asynchronous method call. Any exception that occurred during the execution of an asynchronous method call is rethrown when its future is claimed. But what happens to exceptions of asynchronous method calls, whose futures are not claimed? There are two possible approaches to handle these exceptions. The first is to provide an exception handler for asynchronous method calls [KO02, DUV06]. The second is to use an approach similar to that of Erlang [Arm03], where it is possible to link a process to another process to control errors of linked process.

JCoBox supports the first approach. If one is only interested in exceptions of an asynchronous call, but not in the actual return value, one can register a handler object at the future of the call (cf. Section 4.2.2), which can be used to handle exceptions:

```java
o!start() . when(new Fut.VoidAdapter<Void> {
    public void error(Throwable t) {
        ... // handle t
    });
```

In addition, exceptions that are thrown in an asynchronous method call are always handled by the default exception handler of the cobox in which the call was executed. The default exception handler simply prints the stack trace of the exception to the standard output. It is planned as future work to be able to provide a user-defined exception handler.

**Checked Exceptions.** Java supports checked exceptions, which have to be handled by the caller. Supporting checked exception in asynchronous method calls is cumbersome. If one wants to fully support them, the `await()` and `get()` methods of futures, would have to throw the same checked exceptions as the method that resolves the future. This essentially means that for every method a unique future type would have to be defined. As an alternative, the claiming methods could throw the general `Exception` type. This, however, would require to always handle this exception, even if the called method does not throw checked exceptions. JCoBox adopts the following approach. Checked exceptions are wrapped by the unchecked exception `AsyncException`. The original exception can then be retrieved by using the `getCause()` method. Note that runtime exceptions are not wrapped into `AsyncExceptions`. The methods `await()` and `get()` thus do not throw any checked exceptions.

In addition, there exist variants of the claiming methods ending with `Throw`, which throw the general `Exception` exception. These methods do not wrap checked exceptions, but throw them directly. The following code shows an example.

```java
try {
    someFuture.awaitThrow();
} catch (IOException e) {
    System.out.println("An IOException has been thrown");
} catch (Exception e2) {
```
Object Kind. The exception mechanism transmits values between coboxes. The question is which object kind exceptions should have. JCoBox adapts the approach of Clarke et al. [CWOJ08] and assumes that exceptions are immutable. Exceptions can thus be safely passed to and accessed by any cobox. Classes that inherit direct or indirectly from Exception are implicitly immutable in JCoBox and are treated like immutable classes as described in Section 4.1.8.

4.2.4 Legacy Java Interoperability

For pragmatically reasons, legacy Java classes, i.e., classes not generated by JCoBox, can be used in JCoBox. Sequential legacy Java code is executed in the context of the calling cobox. However, JCoBox offers no concurrency protection for objects of legacy Java classes. Objects of legacy Java classes can be regarded as global objects, not belonging to any cobox. It is not possible to call asynchronous methods on legacy objects.

Using Legacy Java Classes

In general, it is safe to use thread-safe Java classes. For example, using the classes String, Vector, or AtomicInteger, is completely safe in JCoBox. When using thread-unsafe classes like ArrayList, for example, the programmer has to ensure that objects of these classes can never be accessed concurrently by more than one cobox. One way of doing this is to make sure that these objects are never exposed to other coboxes, i.e., are always local to a cobox.

Inheriting from Legacy Java Classes. To inherit from a legacy Java class, the inheriting class has to be annotated with @PlainJava. Classes annotated with @PlainJava are treated like legacy Java classes.

Java Threads

JCoBox works in combination with standard Java threads. It is possible to use legacy code that uses multiple threads internally. It is even possible for legacy code to call back into JCoBox code. In this case, a cobox will act like a reentrant monitor from the perspective of the legacy code. I.e, the legacy code can call back with the original thread, but all other threads are blocked. However, it should be clear that the semantics of such programs will not be easy to understand, as two different concurrency models are combined.
4.2 Extensions

Dynamic Proxies

Legacy Java objects should, in general, not be exposed to other coboxes unless they are thread-safe. Sometimes, however, it is useful to give out such objects. To support these usage scenarios, legacy Java objects can be wrapped by a dynamic proxy object. The proxy object belongs to the cobox, which created it and simply forwards all calls to the target object. The proxy object can then be safely exposed to and used by other coboxes. Proxy objects are created by the method `boxify(Object o, Class<V> c)`, which creates a proxy for object `o` with type `c`. Type `c` must be an interface, which is implemented by the class of `o`. The following example code creates a proxy object for a `LinkedList` object.

```java
List<String> buffer = new LinkedList<String>();
List<String> proxy = boxify(list, List.class);
```

*Implementation Remark.* The JCoBox implementation currently uses Java reflection mechanism to realize dynamic proxy objects. They should thus not be used for performance critical parts of an application.

Swing

Swing [Ora10b] is a toolkit for creating GUIs in Java. Swing is not thread-safe. In particular, all tasks that work with the Swing toolkit must run (in most cases) in the event-dispatching thread of Swing [MW00]. The Swing framework is actually a good example for a typical cobox. At runtime the Swing framework may be regarded as a cobox, which owns all Swing objects. Such a Swing cobox, however, only has a single task. Multiple tasks are not possible and hence tasks are not allowed to suspend or yield.

To make it easier to write correct Swing programs, JCoBox has a special `@Swing` annotation. If a cobox class is annotated with `@Swing`, JCoBox guarantees that all methods, which are asynchronously invoked on objects of the corresponding cobox, are executed by the event-dispatching thread of Swing. This allows for writing standard Swing-based code inside a Swing cobox. In addition, non-GUI coboxes can interact with the Swing cobox in an asynchronous way, leading to a seamless communication of GUI components with other components [SPH10b].

*Implementation Remarks.* The current JCoBox implementation has some limitations, which have to be kept in mind when using a Swing cobox. Inside a Swing cobox it should be avoided to explicitly wait for a future by using `await` or `get`. Instead, only future handlers should be used (cf. Section 4.2.2) as otherwise the Swing event-handling thread may be blocked, resulting in an unresponsive GUI. Another issue is that inside of Swing cobox class constructors no Swing objects should be accessed, as constructors are executed by the thread of the creating cobox (cf. Section 4.2.3).
Chapter 4  The JCoBox Language

@Swing @CoBox class GUI {
    private JFrame frame;
    void init() {
        frame = new JFrame("Hello JCoBox");  // JFrame is a Swing class
        frame.setVisible(true);              // access inside a Swing cobox is safe
    }
}

class SwingApp {
    public static void main(String[] args) {
        setAutomaticShutdown(false);        // the Swing thread may run alone
        new GUI().init();
    }
}

Listing 4.3: A simple “Hello World” example, showing how to work with Swing.

Finally, other coboxes should only interact with objects from Swing coboxes by using asynchronous method calls as synchronous method calls are executed in JCoBox by the thread of the calling cobox (cf. Section 5.5.2).

Example.  Listing 4.3 shows a simple “Hello World” example, which creates a single Swing frame using the Swing cobox GUI. The constructor of GUI asynchronously invokes the init() method, which initializes the Swing frame.

Thread Blocking

When performing any operations that may block a thread, such operations have to be enclosed into startBlocking() and endBlocking() calls, which should be placed into a try-finally block as follows.

    startBlocking();
    try {
        ...  // blocking operations
    } finally { endBlocking(); }

The reason is that tasks in JCoBox are executed by a thread pool (cf. Section 5.3.2). If a thread becomes blocked, it cannot be used for executing other tasks. The startBlocking and endBlocking calls ensure that the size of the thread pool is increased to guarantee enough free threads. Operations that may block a thread include blocking I/O operations and lock acquiring, for example. Note, however, that while a task is blocked it blocks the whole cobox. As, in particular, the latency of I/O operations cannot be predicted, this may not be desired. Consider, for example, a Java legacy class MusicPlayer, which has a play() method to play a song, which blocks the caller until the song has finished. As MusicPlayer is a Java legacy class,
4.2 Extensions

`play()` cannot be invoked asynchronously. A synchronous call, however, blocks the whole cobox while the song is playing. It would, for example, not be possible to execute `stop` messages. For this reason, JCoBox has a mechanism to allow the active task to give up its cobox control, while staying active. The method `releaseCoBox()` gives up control and the method `acquireCoBox()` regains control. These methods should be regarded as mechanisms to escape from the cobox model to interact with legacy code. In the music player example one could write the following code:

```java
MusicPlayer player = ...

public void play() {
    MusicPlayer p = player;
    releaseCoBox();
    try {
        p.play();
    } finally {
        acquireCoBox();
    }
}
```

After giving up control, the thread should not access the state of objects of the cobox anymore until it regained control. Otherwise, data races are possible.

**Signals and Condition Variables**

On top of promises, condition variables can be implemented, which can be used like condition variables known from monitors [Han73, Hoa74], to coordinate the tasks of a cobox. A condition variable is represented by the utility class `Condition`. For example, a bounded buffer can be implemented as follows (only showing the interesting parts):

```java
class BoundedBuffer {
    private final Condition notFull = new Condition();
    private final Condition notEmpty = new Condition();

    void add(Object o) {
        while (isFull())
            notFull.await();
        ...
        notEmpty.signalAll();
    }

    Object remove() {
        while (isEmpty())
            notEmpty.await();
        ...
    }
}
```
Similar to condition variables, **signals** can be used. A signal is a condition variable that can directly pass values between tasks.

```java
class ValueSynchronizer<V> {
    private final Signal<V> nextValue = new Signal<V>();
    V awaitNextValue() { return nextValue.await(); }
    void setNextValue(V v) { nextValue.signalAll(v); }
}
```

In fact, a signal can be regarded as a promise that can be resolved multiple times.

### 4.2.5 Encoding JCoBox Syntax in Standard Java Syntax

The language extension approach taken in this thesis has the advantage that the cobox model can be realized as a seamless language extension, which serves well for experimenting with different language features. However, it has the drawback that many existing Java tools, which work on Java source code, cannot be used for JCoBox programs due to the extended syntax. For a research prototype this is acceptable. However, if the approach should be adopted in industry, tool support is mandatory.

Instead of extending the language with new syntactic constructs, one can also try to create a domain-specific language that uses standard language mechanisms to express the language extension. Most JCoBox constructs are already expressed by using standard Java mechanisms and need no special treatment. However, asynchronous method calls and the extended `new` expression are not standard Java syntax and thus have to be expressed in a different way to be compatible with Java.

#### Encoding Asynchronous Method Calls

Asynchronous method calls in JCoBox have two properties: first, all methods can be invoked asynchronously and synchronously, and second, the result type of an asynchronous method call is a future type of the return type of the called method. This makes it difficult to represent asynchronous method calls in standard Java.

*Calling Mode.* A simple approach is to distinguish the calling mode at the receiver side instead of the caller side. Certain methods are then declared to be asynchronous, in which case calling such methods would result in asynchronous invocations. These methods would then have a declared future type as return type. For example:

```java
@Async Fut<String> m() { ... } // no legal JCoBox code
```
There are several approaches which apply this technique, e.g. Thorn [BFN+09] and Jac [HL06]. However, determining the calling mode at the receiver side has several drawbacks. The main problem is that the programmer of a method has to decide in advance in which mode the method is most likely to be called by clients. In addition, from the perspective of the receiver, it is completely irrelevant in the cobox model whether a method is called synchronously or asynchronously, as both calls are treated as asynchronous calls by the receiver cobox. So it makes not much sense to let the receiver determine the calling mode.

So instead of letting the receiver decide on the calling mode, we use an approach which allows the caller to decide the mode. Expressing an asynchronous call could, for example, be done by using Java comments, e.g., \texttt{x./*!*}/m()\texttt{1}. However, this is not a complete solution as the type of the call is the return type of the method and not a future type. In addition, the comments get lost in the bytecode. Instead, we apply an approach similar to that of [PSH04]. We introduce the marker method, defined in the \texttt{jcobox.Syntax} class,

\begin{verbatim}
  public static <V> V async(V v) { ... }
\end{verbatim}

This method allows receivers to be marked as asynchronous. A call that is invoked on that receiver is executed asynchronously. For example, the asynchronous call in JCoBox source syntax \texttt{x!m()} is expressed as

\begin{verbatim}
  async(x).m()
\end{verbatim}

There also exists the method \texttt{asyncUnordered} to express unordered asynchronous method calls, i.e., it replaces the \texttt{!!} operator.

\textit{Return Type. } The type of an encoded asynchronous method call is the standard return type of the invoked method. The result of this call, however, cannot be used directly. Instead one has to get the future of the call to obtain the result. To get the future one uses the \texttt{getFut(V)} method:

\begin{verbatim}
  public static <V> Fut<V> getFut(V v) { ... }
\end{verbatim}

The \texttt{getFut} call has to be wrapped around an asynchronous call. For example, expressing the JCoBox code \texttt{Fut<String> fut = x!m()} is done by writing:

\begin{verbatim}
  Fut<String> fut = getFut(async(x).m());
\end{verbatim}

As this approach does not work for \texttt{void} methods, because \texttt{void} method calls cannot be used as expressions, there is an additional way for getting a future, namely using the \texttt{getFut} method without a parameter:

\begin{verbatim}
  public static <V> Fut<V> getFut() { ... }
\end{verbatim}

\textsuperscript{1}this syntax is actually supported by JCoBox
Chapter 4 The JCoBox Language

It simply returns the future of the last asynchronous method call. So the above example can be also written as:

```java
async(x).m();
Fut<String> fut = getFut();
```

which works for void methods. The `getFut()` method call must be in the same block, or a nested block of the block, which has the `async` call.

**Checked Exceptions.** The encoding of asynchronous method calls leaves one open problem, namely that checked exceptions cannot be suppressed by the encoding. Consider the call `async(x).m()` and suppose that method `m` is declared to throw a checked exception, e.g., `IOException`. In JCoBox, the call `x.m()`, does not throw any exception, however, the Java compiler does not know this and demands that the checked exception is handled by the caller. Currently, the only solution is to either catch the exception, with the knowledge that it can never be thrown at runtime, or to declare the calling method to throw the exception.

**Encoding the Extended New Expression**

Beside asynchronous method calls, it is required to encode the extended new expression. This is done by a fluent interface [Fow05]. The marker method

```java
public static <V> In<V> create(V v) { ... }
```

is used to get an `In<V>` instance, which only has a single method:

```java
public static class In<V> {
    public V in(Object o) { ... }
}
```

For example, replacing the JCobox expression `new C() in (e)`, is done by writing

```java
create(new C()).in(e);
```

**Discussion**

With the above mechanisms, JCoBox can be completely expressed by standard Java syntax. This means that all existing Java tools can be used for writing JCoBox code. In addition, the encoded syntax is available in bytecode, which makes it possible to implement JCoBox by a bytecode rewriter, which is explained in Section 5.7.2. The only problem are checked exceptions, which require some unpleasant, superfluous code when asynchronously calling methods throw checked exceptions.
4.2.6 Distributed Programming

JCoBox has no built-in distribution features. However, the implementation allows the development of distributed JCoBox programs by using Java's RMI technology [Ora10a]. As the coBox model is well-suited for distributed programming, due to its message-passing model, it is easy to translate an existing JCoBox program into a distributed one, by just using the standard RMI techniques. This section explains how to do this by means of an example.

Example: Distributed Time Server

As an example we show how to implement a distributed time server with JCoBox. We start with a non-distributed implementation shown in Listing 4.4. It consists of the three classes TimeServer, Client, and Main. TimeServer has only a single method getTime, which returns the current time in nanoseconds. Client has a method start, which requests the time from the time server ten times, each time printing it to the command line and waiting for one second between each request. The main method of class Main, initializes the system and calls the start method of the client object.

Using RMI. To make this example distributable using RMI, the following steps have to be taken, which are all standard steps, also needed to make a standard Java application distributable:

- An interface TimeServer for the time server is created, which extends the java.rmi.Remote interface.
- Method getTime must be added to this new interface and it must be declared to throw java.rmi.RemoteException.
- The TimeServer class is renamed to TimeServerImpl and implements the interface TimeServer.
@CoBox class Client {
    void start(TimeServer server) {
        ... // same as above
    }

    public static void main(String[] args) {
        TimeServer server = (TimeServer) Naming.lookup("TimeServer");
        new Client().start(server);
    }
}

interface TimeServer extends Remote {
    long getTime() throws RemoteException;
}

@CoBox class TimeServerImpl implements TimeServer {
    public long getTime() throws RemoteException {
        return System.nanoTime();
    }

    public static void main(String[] args) throws Exception {
        setAutomaticShutdown(false);
        LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
        TimeServer stub = (TimeServer)
            UnicastRemoteObject.exportObject(new TimeServerImpl(), 0);
        Naming.rebind("TimeServer", stub);
    }
}

Listing 4.5: Distributed time server implementation

- An instance of TimeServerImpl must be exported.

- The TimeServerImpl instance must be bound to a name.

On the client side, a main method must be introduced, to be able to start the client independently from the server. In addition, the server instance must be found using the naming service of RMI. The code of the new implementation is shown in Listing 4.5. Important to note is that the implementation of the start method of the client is completely unchanged in the distributed setting. Asynchronous calls as well as futures can be used in the distributed setting like in the non-distributed setting.
4.3 Related Work and Discussion

JCoBox is a language that realizes the cobox model. Several decisions have to be made when designing a language, for example, whether an existing language should be extended or a complete new language should be developed and whether the language should be statically typed or dynamically typed. If a language is extended, one has to decide whether the syntax should be extended, a compiler extension is required, or whether a library is sufficient.

There exist several approaches that realize actor-like languages. Erlang [Arm03], for example, is a complete language of its own; Clojure [Clo09] defines a Lisp dialect that compiles to the JVM; Scala Actors [HO09] is a library for Scala [Sca10], which in turn compiles to the JVM; SALSA [VA01] defines its own language and uses a compiler that generates Java code; ActorFoundry [Act10a] and Kilim [SM08] use a library and bytecode rewriting to realize actors; finally, Actorom [Act10b] is a pure library-based approach for Java.

JCoBox is realized as a Java extension. Java is used as a basis so that it is easier for programmers that are used to Java to adopt JCoBox. In addition, having a robust, existing language basis allows us to concentrate on the unique aspects of the cobox model. Being a nearly conservative Java extension, JCoBox can be used to write standard sequential Java programs which are semantically equivalent to Java. This makes it possible to easily migrate existing sequential Java programs to JCoBox.

JCoBox provides its own syntax, but also has a Java-encoded syntax. Having a specialized syntax provides a better “feel” for the language and is better for experimenting with language extensions. In practice, however, the extended syntax prevents existing Java source tools to work with JCoBox. With the Java-encoded syntax these tools can be reused for JCoBox, which is crucial for writing and maintaining larger JCoBox applications.

Even though, the syntax of JCoBox can be expressed by standard Java syntax, realizing JCoBox as a pure library is not possible. The reason is that JCoBox requires static checks, for example to guarantee immutability. In addition, JCoBox requires to treat synchronous calls and field accesses in a special way to guarantee conformance to the cobox model. This is not possible to achieve in a pure library approach, even not in languages like Scala, which provide better language extension mechanisms than Java [Wor09b].

4.3.1 Actor and Active Object Realizations

This subsection briefly reviews existing realizations of the actor and active object model, which are not already covered by Section 2.5.
ProActive

ProActive [BBC+06] is a framework and middleware for writing parallel and distributed Java programs. The concurrency model of ProActive is based on ASP [CH05]. In addition, ProActive has a component model that is based on Fractal [BCM03]. ProActive is realized as a library and uses runtime bytecode rewriting.

Scala Actors

Scala Actors [HO09] is an actor library for Scala [Sca10], which is very similar to Erlang, but runs on the JVM. Even though it is realized as a library, it integrates well into Scala, benefiting from several Scala features not present in Java. Communication is based on messages, which are standard Scala objects. They support pattern matching, futures and join patterns [HC07]. Actors in this model are single objects, thus multiple service objects are not supported. Also not supported are multiple cooperating tasks within a single actor. The communication mechanism is orthogonal and incompatible to method calls, i.e., calling methods on actors can lead to data races.

ActorFoundry

ActorFoundry [KSA09, Act10a], is an actor implementation for Java. Messages are send by using strings, which specify method names. Hence, message sending is not statically typed. The implementation of ActorFoundry is based on Kilim [SM08] to support lightweight continuations. Messages can either be send by copy, using Java’s serialization mechanism, or using an unchecked reference copying.

Actorom

Actorom [Act10b] is an actor implementation for Java, which features so-called topologies. Topologies are execution environments for actors. Topologies can either be local or distributed, supporting local and distributed actors, respectively. Topologies are an address space, specify the threading model for its actors, and specify how to handle actor failures. Messages in Actorom are instances of standard Java objects. Method annotations specify with type-based patterns how to react to messages. The communication mechanism is not compatible with standard method calls.

Clojure

Clojure [Clo09] is a Lisp-dialect that compiles to JVM bytecode. Clojure is a functional language and thus state is in general immutable. It supports concurrency by threads and asynchronous executions of functions by so-called agents. Safe state sharing is provided by mutable references that are protected by a software transactional memory system (STM).
4.3.2 Discussion, Limitations, and Future Work

This subsection discusses the JCoBox language and its limitations, and potential extensions and future directions.

Message and Task Scheduling

Currently, a cobox serves all method calls it gets in the same order as they appear. There is no direct mechanism to define a certain communication protocol. Instead, a programmer has to use promises or monitor-like condition variables. There exist several different solutions for scheduling of messages and tasks. One are guards at method declarations [HL06] or at suspension points [dBCJ07]. Join patterns [FG96, BCF02] could also be used. An actor like mechanism, where a body explicitly specifies a communication protocol [Ame89, ACE+88, CHS04, KGMO04, BFN+09] could be integrated into JCoBox. Finally, one may specify the scheduling of messages in a separate object as done in SOM [CMET04] and POM [CMPET08]. This is already partly supported by JCoBox as the scheduler object is independent from the cobox object, a feature which is used to implement the Swing support.

Message Priorities

Some active object languages, e.g. ABCL/1 [Yon90] and Thorn [BFN+09] have special messages with higher priority then others. This is sometimes useful, especially to stop activities. ABCL/1 has express messages, which interrupt the execution of an active object. This has the great disadvantage that tasks do not have exclusive access anymore and may get interrupted at anytime. ABCL/1 therefore introduces an atomic construct, which guarantees atomic execution. Thorn allows messages with different priorities. The message with the highest priority is selected from the mailbox. However, high-priority messages cannot interrupt running executions, thus do not have the problems of ABCL/1. Messages with different priorities, like in Thorn, are a straightforward extension to the cobox model.

Data Transfer

Currently, data in JCoBox is transferred between coboxes either by copying or by using a simple form of class immutability. The current form of immutability is very strict. More flexible immutability notions [HP09, CWOJ08, HPSS07] could be used in JCoBox instead.

Sometimes neither copying nor immutability is an option. A network connection object, for example, can neither be immutable nor can be copied. In addition, copying large mutable data structures, like image buffers, imposes a high performance overhead. It is thus desirable to be able to safely transfer mutable objects by reference. Several solutions to this problem exist that are based on unique references [CW03,
CWOJ08, SM08, FAH+06]. These approaches could be added to JCoBox to support the move of data between coboxes without the copying overhead.

**Near and Far Objects**

Near and far objects cannot be distinguished statically in JCoBox. This means that when reading a JCoBox program it might be difficult to understand the cobox structure that a programmer had in mind. A simple variant of an ownership type system [CWOJ08] can be developed that makes these object roles statically clear. This information can be used for several purposes. It allows the compiler to apply optimizations for near calls. It can help with verification of coboxes as near calls do not leave the cobox boundary and thus do not participate in the externally visible behavior. Finally, it can help the programmer to better document and understand the structure of JCoBox programs.
The implementation of JCoBox consists of two parts: a runtime library and a compiler. As it is much easier to maintain a library than to maintain a compiler, the implementation tries to do as much as possible in the library. The compiler mainly generates code that delegates into the runtime library. The runtime library and the generated code are thus tightly coupled. JCoBox has two compiler implementations. One that takes JCoBox source files and generates Java code, and one that rewrites JVM bytecode that has been generated by a Java compiler from Java source files that use the Java-encoded JCoBox syntax (cf. Section 4.2.5).

5.1 Requirements

Before we discuss possible implementation strategies, we give a set of requirements which the JCoBox implementation must respect. Some of these requirements are given by the definition of the JCoBox language in the previous chapter.

1. JCoBox must be implemented on top of Java. In particular, the sequential programming model of Java should be preserved.

2. JCoBox must be safe, i.e., data races must not be possible as long as only JCoBox mechanisms are used.

3. The additional JCoBox mechanisms should be strongly-typed. Especially, asynchronous method calls must be implemented in a type-safe way.

4. The additional static JCoBox constraints must be checked.

5. Methods and fields of objects should be accessible in a synchronous way, i.e., by using the standard dot-notation of Java, even if they belong to different coboxes. Such accesses must be equivalent to asynchronous method calls with an immediate \texttt{get()} on the resulting future.
6. It should be possible that JCoBox code can interact in a thread-safe way with legacy Java code and vice versa.

Besides these functional requirements, the JCoBox implementation must also meet certain performance requirements, described in the following.

### 5.1.1 Performance Requirements

JCoBox is a general purpose programming language. This means that it should support a wide range of possible usage scenarios. As the characteristics of possible applications can vary in many aspects, it is necessary that JCoBox is globally optimized as much as possible. The following aspects are the most relevant.

1. The total number of simultaneously existing coboxes.
2. The total number of simultaneously existing unhandled messages.
3. The total number of simultaneously existing tasks.
4. The time it takes to send and receive messages.
5. The time it takes to create tasks.
6. The time it takes to cooperatively schedule tasks.
7. The utilization of multiple cores.

As an evaluation criteria, JCoBox should be comparable in these aspects with other actor implementations for the JVM, e.g., Scala Actors [HO09]. For example, it should be possible to comfortably have millions of simultaneously existing coboxes.

### 5.2 Possible Implementation Options

Implementing a concurrency model on top of an existing language, like Java, can essentially be done in four ways (see also [BGL98]):

1. as a library,
2. by using runtime reflection,
3. by bytecode rewriting (at compile time or runtime),
4. by a compiler extension.

Each of these approaches has advantages and disadvantages, which we briefly discuss in the context of JCoBox and Java. The first three approaches only work with the Java-encoded syntax proposed in Section 4.2.5. It is not possible to have an extended syntax with these approaches. Having a Java-compatible syntax, however, has the advantage that the standard Java tools can be used to write JCoBox programs. Especially, tools that work on Java source code can be completely reused. A compiler
5.3 Realizing the CoBox Model in a Thread-Based Model

extension, on the other hand, allows for arbitrary changing the Java syntax and is the only way to implement the extended syntax.

Only relying on a library has the disadvantage that the correct usage of language extension cannot be guaranteed, e.g., data races may still be possible. In addition, certain language extensions, like asynchronous method calls for example, are difficult or impossible to implement as a library. Finally, the semantics of standard synchronous method calls cannot be modified by a library. A library, however, is the least invasive approach and is thus most likely to be accepted by most programmers.

Compared to a pure library approach, runtime reflection allows for modifying the semantics of synchronous method calls. It can also be used to realize asynchronous method calls in a type-safe way to a certain extend. But runtime reflection has, in general, a high performance overhead and it would most likely not be possible to meet the above mentioned performance requirements.

Runtime bytecode rewriting is similar to runtime reflection, but does not have the performance penalties. It is thus a viable implementation option. Using only runtime mechanisms, however, does not allow for statically checking certain conditions, like immutability constraints. Runtime bytecode rewriting approaches are broadly used in practice and would thus be likely be accepted most programmers. Except for the extended syntax, a compile time bytecode rewriting approach could completely realize the above mentioned requirements.

This thesis implements JCoBox as a compiler extension as well as by using static bytecode rewriting. The compiler extension supports the complete JCoBox language as described in the previous chapter. It supports the extended syntax as well as the Java-encoded syntax. The bytecode rewriter only supports the extended syntax. The compiler extension has the advantage that it is easy to integrate and experiment with new language constructs, but it has the disadvantage that the full Java language has to be supported. The bytecode rewriter, on the other hand, must only support JVM bytecode, which is much simpler, but realizing new language constructs is more difficult.

5.3 Realizing the CoBox Model in a Thread-Based Model

This section discusses possible implementation strategies for realizing the cobox model on top of a standard object-oriented programming model based on threads. There are two crucial questions that have to be answered. The first is how to realize the relation between objects and coboxes and the second is, how to map tasks to threads.
5.3.1 The Object–CoBox Relation

The first question is how the relation between objects and coboxes should be implemented. This can be done either by using a global map, which maps each object to its cobox, or by adding a field to each object referring to its cobox. The first solution has the advantage that classes do not have to be changed. But it has a critical disadvantage: the map is a large scalability bottleneck as the relation must be checked very often and in parallel. If, instead, each object has a field referring to its cobox, reading that field is very fast and scalable.

5.3.2 How and When to Use Threads

One of the most important design decisions for an actor implementation is how and when to use threads. Threads are, in general, expensive resources and should be used with care.

One Thread per CoBox

One way to implement the cobox model is to assign one thread to every cobox. This thread would realize an execution loop that executes all tasks of the cobox, one by one. Such an implementation is, for example, used in SALSA [VA01]. This approach has two disadvantages. The first one is that it is very expensive in terms of resources, because every cobox requires a separate thread, even if it has no tasks at all. This bounds the number of coboxes to the maximal number of threads, which is in general very limited on typical systems [KSA09]. The second problem is that JCoBox allows tasks to suspend or to yield at arbitrary code positions. This means that the current execution stack of these tasks has to be remembered, while other tasks are executed. Java has no built-in mechanisms to store the current execution stack, except by using the underlying thread itself for this. Hence, in order to be able to suspend or yield a task, the underlying thread itself has to be suspended. To execute other tasks in the cobox a fresh execution thread is required. But this even increases the number of threads that are required for a single cobox.

One Thread per Task

Another straightforward implementation strategy is to assign a thread to every task. As soon as an asynchronous method is invoked, a task is created and assigned to a thread that executes it. Each cobox then has a single lock, which has to be acquired by a thread before it can execute the task. After it finishes the execution of the task, the thread terminates. This implementation has the advantage that thread suspension is easily implemented by simply giving up the cobox lock when the task suspends and acquiring it again when the task resumes. In addition, coboxes are cheap because they do not require a thread when they are idle. However, tasks now become very
5.3 Realizing the CoBox Model in a Thread-Based Model

expensive. Tasks do even require a thread when they have not been executed yet, which means that each message effectively requires a separate thread.

**Using a Thread Pool**

To reduce the number of threads, thread pools can be used. A thread pool manages a number of worker threads to be used for executing tasks. Thread pools reduce the total number of running threads and reduce the cost for creating threads, as threads are in general reused. A thread pool has an internal task queue, where tasks are buffered for eventual execution. The worker threads take new tasks from that queue and execute them.

So instead of immediately assigning a thread to a task, the task is given to a thread pool instead. This potentially reduces the number of necessary threads. However, tasks may still be executed by a thread, even if the target cobox is already locked by some other task. In that case the worker thread is blocked and the size of the thread pool must be increased to guarantee progress. In the worst case, every task has a separate thread, where most of the threads are blocked.

**One Thread per Activated Task**

To avoid that a thread executes a task that may be blocked, one can implement a way to ensure that a task is only executed by a thread if it is guaranteed that the task can really be executed.

One way of achieving this is as follows. Every cobox gets its own task queue. New tasks are first added to that queue. If a task reaches the head of the queue it gets the cobox lock and is added to the global thread pool for execution. When the task is eventually executed, it already has the lock and is not blocked. With this implementation strategy, a task only has an assigned thread after it has already been activated. If it was not activated yet, it will not require a thread. However, if a task suspends or yields, it must still keep its thread to store the current execution stack. A suspended task still requires a separate thread. Figure 5.1 illustrates this execution strategy. This means that the number of suspended and yielded tasks is limited by the maximal number of threads of the underlying system.

**Optimizing Task Suspension**

To further reduce the number of threads one can reuse suspended or blocked threads. A task in JCoBox suspends or blocks, in general, when it waits for a future. For example, the expression \( x! m().\text{await}() \), asynchronously invokes \( m() \) on \( x \) and then suspends the task until the resulting future is resolved. This future, however, is only resolved after method \( m() \) has been executed. As the executing thread can only be resumed after method \( m() \) has been executed, it can just as well execute \( m() \) itself. It is similar when a future is claimed by using \( \text{get}() \) as in this case the thread is blocked until the corresponding method has been executed. With this technique it is possible
to reuse suspended threads in many cases. However, it might be the case that the invoked method is executed by a different thread already, which then prevents the reuse.

5.4 The JCoBox Runtime

After having explained the main design problems of the JCoBox implementation with respect to a thread-based concurrency model, this section explains the main design and implementation aspects of the JCoBox runtime library.

5.4.1 JCoBox Runtime Access

The JCoBox runtime is realized by classes, that implement the JCoBoxRuntime interface. JCoBox-generated code accesses the runtime using static methods of the JCoBoxCore class (see Figure 5.2). It has a single static final field, which refers to an instance of JCoBoxRuntime. All methods of the JCoBoxCore class simply forward to that instance. This design allows for easily testing different runtime implementations, without changing the compiler.

Figure 5.1: The execution strategy of tasks. Suspected tasks always require a separate thread, ready tasks that have not been executed yet, do not require a thread.
5.4.2 Standard Objects

Standard objects in JCoBox are instances of standard classes. Every standard class inherits from CoObjectClass (see Figure 5.3). This is realized by declaring every standard class without an explicit super class to extend CoObjectClass. Transfer, immutable and classes inheriting from legacy Java classes do not inherit from CoObjectClass. The CoObjectClass has a single final field referring to the owning cobox and it implements the StandardObject interface, which is a marker interface to indicate standard classes. If a class is annotated with @CoBox the generated class, in addition, implements the marker interface CoBoxClass.

5.4.3 CoBoxes

CoBoxes are represented at runtime by instances of the CoBox interface. The CoBoxImpl class implements this interface (see Figure 5.4). It has only a single field, which refers to the scheduler that is responsible for scheduling the tasks of the cobox.
5.4.4 Asynchronous Calls

Asynchronous calls are represented at runtime by instances of the `AsyncCall<T,R>` interface. The type parameter `T` is the type of the target of the called method, and the type parameter `R` represents the methods return type (see Figure 5.5). The interface implements the `java.lang.Runnable` interface, which allows it to be executed by a standard thread pool. The interface is implemented by the `AsyncCallImpl` class. This class has two fields, one referring to the target of the call, and one referring to the cobox of the target object. There exist two subclasses of this class. The `FutureCall` class represents an asynchronous call, with an associated promise that is resolved by the call. The `OneWayCall` class is an optimized version, which is used for asynchronous calls, where the resulting future is not used by the caller.

![Figure 5.5: The AsyncCall interface and its implementing classes.](image)

5.4.5 Futures and Promises

Figure 5.6 shows the class diagram of futures and promises. Futures and promises are instances of the interfaces `Fut<V>` and `Promise<V>` respectively. The type parameter `V` represents in both cases the type of their potential value. Both, futures and promises, are implemented by inheriting from the `ValueHolder` class. The `ValueHolder` class efficiently implements the storage of an optional value and an exception. Futures are implemented by the `AbstractFuture` class, which inherits from `ValueHolder`. Futures that have an associated promise, are instances of the `PromiseFuture` class, which has a reference to its promise. The `PromiseImpl` class implements a promise. It has an arbitrary number of associated futures. For promises that are resolved by an asynchronous method call the additional classes `CallPromise` and `CallFuture` are used.
5.4 The JCoBox Runtime

Figure 5.6: Class diagram, showing the implementation of futures and promises.

5.4.6 Threads

One question that has to be solved is, how to find out the current cobox, i.e., the cobox of the currently executing task. One way to achieve this is to extend every method with an implicit parameter that represents the current cobox. Changing the signature of methods, however, leads to problems when using legacy Java code. It would not be possible, for example, to implement a legacy Java interface, which does not have such a parameter. Instead of adding a parameter to each method, it is also possible to store the current cobox in a thread-local field. Thread-local fields in Java, however, are implemented internally by a hash map. Hence, accessing such a field has some performance overhead. Except the main thread and threads created by legacy Java thread, all threads are created by the JCoBox runtime. This allows us to define a special thread class that has an additional field referring to the current cobox. The class shown in Figure 5.7 is the thread class that is used by the JCoBox runtime. As not all threads can be created by the JCoBox runtime, thread-local fields are used for all threads that are not instances of the CoBoxThreadImpl class.

Figure 5.7: Class diagram of the CoBoxThreadImpl class.
5.4.7 The CoBox Scheduler

A cobox uses a scheduler object that handles the scheduling of tasks. This subsection describes the implementation of the QueueScheduler3 class, which is the default scheduler used in JCoBox (see Figure 5.8). The scheduler has an internal array-based queue that holds all tasks of the cobox. The head of that queue represents the currently active task. A task can either be an AsyncCall object, if it originated from an asynchronous method call, or it can be a Thread object, if it originated from a synchronous call. The scheduler executes the tasks in FIFO order. Asynchronous tasks are executed by a TaskExecutor object, which is passed to a thread pool. The additional object is needed as after a task has been executed, the executing thread is responsible for scheduling the next task, which is done by the TaskExecutor. When a synchronous call tries to enter the scheduler, it is added to the task queue and blocked until it reaches the head of the task queue. Managing synchronous calls and asynchronous calls in the same queue ensures that the ordering of method calls is guaranteed. So the code `x!m(); x.n();` is executed by first executing `m()` and then `n()` if `x` is a far reference.

The whole scheduling process happens without the need of an additional scheduler thread. This is achieved by letting the active task of a cobox execute the scheduling code before it terminates. So a cobox without any running task does not require a separate thread.

![Class diagram of the QueueScheduler3 class.](image)
5.4.8 The Thread Pool

The class `ExecutorBasedThreadPool` implements the default thread pool of JCoBox (see Figure 5.9). It internally uses the `ThreadPoolExecutor` class from the JDK package `java.util.concurrent`. To guarantee that there are always enough running threads in the thread pool, the size of the pool is dynamically updated. Whenever a task blocks, it informs the thread pool about this, which internally modifies a size delta. A separate thread executes the `PoolSizeUpdater`, which constantly checks the size delta and modifies the size of the thread pool if required. In addition, another thread constantly monitors task activity and adds additional threads to the pool when needed, to ensure liveness.

```
Figure 5.9: Class diagram of the ExecutorBasedThreadPool class.
```

5.5 Java Translation

The runtime library is used at runtime by the code that is generated by the JCoBox compiler. This section explains how JCoBox code is translated into Java code.

5.5.1 Asynchronous Method Calls

As explained in Section 5.4.4, asynchronous method calls are represented by the `AsyncCall` interface. Concrete implementations are the classes `FutureCall` and `OneWayCall`. These classes only implement common code for asynchronous calls. The code for each individual call is implemented in a generated class that subclasses one of these two classes. Which one is subclassed depends on whether the future is needed or not. If it is statically clear that the future is not needed, the `OneWayCall` class is extended, otherwise the `FutureCall` class. The generated class is responsible for holding and copying the arguments of the call as well as for invoking the corresponding method synchronously.

Listing 5.1 shows an example translation from an asynchronous method call in JCoBox to the corresponding Java code. For the asynchronous method call, a class
B\_n\_0 is created, which extends class OneWayCall\(\langle B, R \rangle\), where type argument \(B\) corresponds to the target type of the call, and \(R\) to the return type. Class B\_n\_0 is an inner class of class A to avoid accidental name collisions for separately generated code parts. All code inside class A, including code of inner classes of A, share these generated classes, which means that for identical asynchronous calls only one class is generated. A disadvantage of this generation scheme is that for code outside of A, duplicated classes are generated for identical asynchronous calls, i.e., calls targeting the same method.

The actual asynchronous call \(b \! \# n(t)\) is translated as follows (Line 7). First an instance of class B\_n\_0 is created, where the target \(b\) is passed to the constructor. The target is then passed by the constructor to the super constructor (Line 14), which stores it in a protected field \(\text{target}\). On the new object the method \(\text{init}(t)\) is invoked. Method \(\text{init}\) copies all method arguments and stores them in internal fields (Line 25). This means that the copying of the arguments happens by the invoking task, so that the state of copied objects reflects the state at the time of the issuing of the asynchronous call. Finally, the new call object is passed to the oneWayCall method, which passes the object to the JCoBox runtime for an eventual execution. When the call is finally executed, the call method of the call object is invoked, which just invokes the original method \(n\) on the target object and passes the arguments, stored in the fields of the call object (Line 18). This call is then executed in the context of the corresponding target cobox.

### 5.5.2 Synchronous Method Calls

A synchronous call in JCoBox can either be a near call or a far call. A near call must be directly executed by the invoking task, where a far call must be semantically equivalent to an asynchronous call with an immediate \(\text{get}()\) (cf. Section 4.1.4). A straightforward way to implement this is to introduce a check that tests whether the target object is a near object or a far object. In the former case the method is directly invoked, and in the latter case the call is simply replaced by an asynchronous call with an immediate \(\text{get}()\). Whereas this approach is sufficient, it is not very efficient, because an asynchronous method call requires the creation of a new object.

Instead of implementing it this way, JCoBox uses a different approach. As the calling thread has to wait until the called method has finished anyway, it can just as well execute the method itself. To ensure that these calls are semantically correct, the thread does not invoke the original method directly, but instead a wrapper method, which is created for every method of standard classes and cobox classes. An example of a generated wrapper method is shown in Listing 5.2. The wrapper method actually has the same name as the original method, and the original method is renamed by appending \$orig to the name. First, the target cobox is determined by getting the cobox of the \textit{this} object (Line 3). The source cobox is found out by getting the current cobox, which is stored in the current thread object (Line 4). It is then checked whether source and target coboxes are different (Line 5). This check is cheap as it
5.5 Java Translation

JCoBox Code

```java
class A {
    void m(B b, T t) {
        b!n(t);
    }
}
```

Java Code

```java
class A extends CoObjectClass {
    void m(B b, T t) { /* see Listing 5.2 */ }
    void m$orig(B b, T t) {
        // an asynchronous call is replaced by the instantiation of the
        // corresponding generated class. That object is then
        // passed to the JCoBox runtime for scheduling.
        JCoBoxCore.oneWayCall(new A.B_n_0(b).init(t));
    }

    // For each (unique) asynchronous call a class is created.
    // Here the class extends OneWayCall as the future is not needed
    static final class B_n_0 extends OneWayCall<B, R> {
        private T arg0; // holds copy of method argument
        B_n_0(B target) { super(target); } // initialize target field

        // call is later invoked by the executing thread
        public R call() throws Exception {
            return this.target.n(arg0); // invoke original method
        }

        // copies the arguments and stores them in internal fields
        public OneWayCall<B, R> init(T p0) {
            CoBox cb = JCoBoxCore.currentCoBox();
            CoBox tb = JCoBoxCore.getCoBoxOf(this.target);
            arg0 = TransferUtil.copyTo(cb,p0,tb);
            return this;
        }
    }
}
```

Listing 5.1: Translation of an asynchronous method call in JCoBox to the corresponding
Java code.
does not require any synchronization. This is because the cobox field of the target object is \texttt{final} and the current cobox is stored in a field of the thread object, which is guaranteed to be only accessed by the thread itself. If the coboxes are identical, the call is actually a near call and thus the method can be directly executed (Line 6). If the coboxes are different, the method arguments have to be copied (Line 8 and Line 9). The thread then enters the target cobox, which essentially means that it is added to the scheduler of the target cobox, which is explained in Section 5.4.7. After the thread has successfully entered the target cobox, the original method is invoked with the copied arguments (Line 12). Finally, the thread returns to the source cobox (Line 15). The mechanism of generating wrapper methods, also enables a seamless interoperability with legacy Java, as these methods can be safely called by legacy Java threads.

\textit{Synchronous call optimizations.} Synchronous calls with \texttt{this} and \texttt{super} as targets can be directly invoked without checking the cobox of the target objects because it is statically clear that the cobox is not left by these calls. This can be done in any context. JCoBox applies this optimization so that these calls are executed like standard Java calls without any overhead. A similar optimization is also done in Guava [BST00]. Note that such an optimization cannot be generally done in standard Java for calls of synchronized methods, as it is in general statically not known whether the object lock is already held or not.

5.5.3 Object Creation

One critical aspect of the creation of objects is when and how to set the cobox of an object. The problem is that code inside the constructor of an object must be executed in the correct cobox. For standard classes this is no problem as the cobox is not changed when they are created. But it becomes difficult when the class is a cobox class, or if the extended \texttt{new} expression is used.

The straightforward way is to extend every object constructor with an additional parameter that specifies the cobox. This parameter is passed up in the constructor hierarchy until reaching the \texttt{CoObjectClass}. The constructor of the \texttt{CoObjectClass} then assigns the cobox to the internal field and the thread can enter the new cobox:

\begin{verbatim}
class CoObjectClass { 
    final CoBoxImpl cobox;
    CoObjectClass(CoBoxImpl cobox) { 
        this.cobox = cobox;
        if (this.isCoBoxClass())
            JCoBoxCore.enterCoBox(cobox);
    }
}
\end{verbatim}

This approach, however, has a problem. Consider the following JCoBox code:
5.5 Java Translation

JCoBox Code

```java
class A {
    void m(B b, T t) {
        b!n(t);
    }
}
```

Java Code

```java
class A extends CoObjectClass {
    void m(B b, T t) {
        CoBoxImpl target = JCoBoxCore.getCoBoxOf(this); // get target cobox
        CoBoxImpl source = JCoBoxCore.currentCoBox(); // get current cobox
        if (target == source) { // check whether cobox has changed
            this.m$orig(b, t); // call original method if cobox has not changed
        } else {
            B b2 = TransferUtil.copyTo(source, b, target); // copy arguments
            T t2 = TransferUtil.copyTo(source, t, target);
            JCoBoxCore.enterCoBox(source, target); // enter target cobox
            try {
                return this.m$orig(b2, t2); // call original method
            } finally {
                // return to source cobox
                JCoBoxCore.returnToCoBox(source, target);
            }
        }
    }

    void m$orig(B b, T t) { /* see Listing 5.1 */ }

    static final class B_n_0 extends OneWayCall<B, R> {
        /* see Listing 5.1 */
    }
}
```

Listing 5.2: Example for the implementation of a wrapper method
@CoBox class A {
    A() { this(new B()); }
    A(B b) { ... }
}

The parameterless constructor of A creates a new instance of B and passes it to another constructor as a parameter. The instance of B, however, is created before the call to the other constructor. This also means that B is created before the constructor of the super class of A is called, i.e., the CoObjectClass constructor. Thus, class B is created in the context of the old cobox and not the new one. Instead of entering the new cobox in the constructor, one can also enter the new cobox before creating the new object. To do this a static method is created for every constructor of a cobox class. Listing 5.3 shows this by means of an example. For the constructor A() the static method A.jcoBox$createA() is generated. The implementation is essentially equal to the wrapper methods that are created for methods. In particular, they also copy constructor arguments if necessary. For standard classes, static factory methods are only created to support the extended new expression. The creation of standard classes by the standard new expression does not require factory methods, only new expressions with cobox classes and all extended new expressions are replaced by calls to the generated factory methods.

Note that the factory methods do only enter the target cobox, they do neither set the cobox field of the newly created object, nor pass the cobox to its constructor. But how is the cobox field of object A set? The answer is that it is set in the CoObjectClass constructor to the current cobox. The actual CoBoxClass constructor now looks as follows:

class CoObjectClass {
    final CoBox cobox;
    CoObjectClass() {
        cobox = JCoBoxCore.currentCoBox();
    }
}

As the constructor is executed in the context of the new cobox, the current cobox is always the cobox of the created object.

## 5.5.4 Transfer Classes

Every transfer class implements the TransferObject interface, which is defined as follows:

```java
public interface TransferObject extends Serializable {
    Object deepCopy(IdentityHashMap<Object,Object> clones);
}
```
For every transfer object the implementation of the `deepCopy` method is automatically generated. The `clones` parameter of the method is used for storing already copied objects, for treating recursive data structures. The implementation is similar to the default implementation of serializable classes, but is much faster, because no intermediate byte stream is used. In the distributed setting, the standard serialization mechanism of RMI is used. Transfer classes are otherwise used and accessed without further protection. In particular, the compiler does not create wrapper methods for transfer classes. Thus, accessing transfer objects happens without any overhead. The reason for this is simple. As JCoBox guarantees that transfer objects are always local to a cobox, access to this objects can only happen from one cobox. Transfer classes also do not have a cobox field. Tracking the cobox relation is not necessary for transfer objects, as the cobox of a transfer object is always the current cobox.

### 5.5.5 Immutable Classes

Immutable classes are similar to transfer classes in the sense that they do not have a field referring to their owning cobox and that no wrapper methods are created for immutable classes. The reason for this is different, though. Immutable classes
have no cobox field, because immutable objects do not belong to a any cobox. Wrapper methods are not needed, because access to immutable objects is always thread-safe. Immutability is guaranteed by making all fields of immutable classes final. To prevent that immutable objects refer to transfer objects, the compiler checks that the right hand side of assignments to fields of immutable classes are not a subtype of TransferObject. If the right hand side is of type Object or of an unspecific interface (cf. Section 4.2.3), a dynamic check is generated that throws an ImmutabilityException if the object at runtime is a subtype of TransferObject. In addition, the use of futures in immutable classes is limited, as futures to transfer objects are also not allowed. Thus, raw future types are not allowed, as well as future types that have a subtype of TransferObject as parameter type. In addition, future types may not have corresponding nested future types. For example, the following future types are not allowed, assuming that type A is a transfer class: Fut, Fut<A>, Fut<Fut<A>>.

5.6 Extensions

This section explains two extensions of the JCoBox runtime, namely deadlock detection and RMI support.

5.6.1 Deadlock Detection

JCoBox supports a runtime deadlock detection, which is useful for debugging. It is turned off by default, but can be turned on by starting a JCoBox program with the Java command line parameter -Djcobox.deadlockdetection=true. The deadlock detection mechanism of JCoBox detects deadlocks, where the length of the wait-for chain of tasks is either one or two. This covers the most cases in practice and can be implemented efficiently. If a deadlock is detected, a DeadlockException exception is thrown. The deadlock detection is implemented by using a different cobox implementation when deadlock detection is enabled. This implementation inherits from CoBoxImpl and extends it with two additional fields: one for holding a promise reference and one for holding a cobox reference. Both fields are set when the active task of the cobox exclusively waits for a future. In that case the promise field is set to the promise of the future. If the promise is a CallPromise (cf. Section 5.4.5), the cobox field is set to the cobox of the FutureCall object that resolves that promise. In addition, every thread gets an additional field holding a promise. That field is set to the promise of the call that resolves the promise and that is currently executed by the thread. Whenever a task now calls await or get on a future, the deadlock detection mechanism tries to find out, using the additional fields, whether there is a wait-for chain of length one or two.
Reentrancy Exceptions. The deadlock detection explained above only addresses asynchronous method calls with futures. As JCoBox also supports synchronous method calls, which are executed without the use of futures, an additional mechanism is implemented to handle these calls. There is a special category of deadlocks, which happen due to synchronous reentrant calls. Figure 5.10 shows an example. It consists of two cobox classes A and B, which mutually call each other. Now consider the expression `new A().m(new B())`. The sequence diagram of Figure 5.10 illustrates what happens at runtime. Method `m()` of class A synchronously calls method `n()` of class B, which immediately calls back. As the cobox of A is blocked, a deadlock occurs. The deadlock detection of JCoBox detects all deadlocks that occur due to such reentrant calls, independent of the number of intermediate calls. In such situations a ReentrancyException exception is thrown instead of a DeadlockException, to give the programmer more details about the reason of the deadlock.

Figure 5.10: Example code and sequence diagram showing a reentrant synchronous call that leads to a reentrancy exception.

5.6.2 RMI Support

The JCoBox implementation has built-in RMI [Ora10a] support for writing distributed applications. It is only a proof-of-concept implementation, which can be improved in several aspects. This subsection presents some important design decisions.

JCoBox relies on the standard RMI mechanisms for its implementation. To make objects remotely accessible, their classes must implement an interface which inherits from `java.rmi.Remote`. The objects must then be exported with standard RMI mechanisms. As standard objects in the cobox model should be remotely accessible, the class `CoObjectClass` implements the `RemoteCoObject` interface, which is shown in Listing 5.4.

CoBoxes themselves are never accessed remotely. All interaction happens only via the remote objects of a cobox. Asynchronous method calls are realized by
Chapter 5 Implementation

interface RemoteCoObject extends Remote {
    void remoteScheduleTask(AsyncCall<?, ?> task) throws RemoteException;
    CoBox.ID remoteGetCoBoxID() throws RemoteException;
}

Listing 5.4: The RemoteCoObject interface

creating the corresponding AsyncCall object (cf. Section 5.4.4) and passing it to the remoteScheduleTask method, which then adds it to the cobox of the object. The AsyncCall interface implements the Serializable interface and instances are thus serialized when transmitted over the network. Different coboxes are distinguished by a globally unique\(^1\) cobox identifier CoBox.ID.

The RMI Scheduler

To decouple network access from calling code, there is an RMIScheduler class. Task objects of asynchronous method calls are not directly passed to the receiver object, but are given to the RMI scheduler. The RMI scheduler has its own thread pool for executing remote methods. In addition, it guarantees the partial ordering of asynchronous method calls by using the unique cobox identifier of the target object, which has to be obtained from the target object before issuing the first call. In addition to that, the RMI scheduler catches RemoteExceptions and delegates them to registered exception handlers.

Promises and Futures

The remote implementation of promises and futures appeared to be more difficult than expected at first sight. For promises it is clear that they must be remotely accessible as they can be shared between coboxes. Futures should be copied between coboxes, but they cannot use the default object serialization. The reason is twofold. First, futures should not be completely serialized because if they are resolved to a transfer object, the transfer object should not be copied (cf. Table 3.10, page 59). Instead, only the promise reference should be copied and the new future should be in an unresolved state. Second, futures are resolved by promises, which might reside on a different machine. Thus, futures must be remotely accessible by promises.

To solve the first issue, futures implement the Serializable interface, but provide a special serialization proxy [Blo08] instead of relying on the standard serialization. A serialization proxy is an object that is serialized in place of the actual object in question. The serialization proxy of a future only has a reference to the promise of the future. After the proxy has been copied via the network, it creates a new future instance and initializes it with its promise. The future then registers itself at the

\(^{\text{1}}\)The global uniqueness is ensured by using an Universally Unique Identifier (UUID) [UUID08]
promise. However, as the future itself cannot be remotely accessed as it is serialized, the future actually registers a remote proxy object at the promise. So the promise does not directly resolve the future, but resolves the proxy object, which in turn resolves the future. With these mechanisms a future is neither directly transferred over the network nor directly accessed remotely.

5.7 Translation Tools

We implemented two tools to translate JCoBox code into Java code: a compiler that takes JCoBox source code as input and a bytecode rewriter that takes Java bytecode as input.

5.7.1 The JCoBox Compiler

The JCoBox Compiler (JCoBoxC) translates JCoBox source code into standard Java source code. JCoBoxC is implemented as an extension for the compiler framework Polyglot 1.3.5 [Pol10, NCM03]. Support for Java 5 features, e.g., generics, is realized by the JL5 extension [JL510]. The Java code that is generated by JCoBoxC is compiled to JVM bytecode by a standard Java compiler (see Figure 5.11). JCoBoxC also understands the Java-encoded JCoBox syntax described in Section 4.2.5.

To accept JCoBox syntax, the compiler extends the JL5 grammar to support asynchronous method calls as well as the extended new expression. The only difficulty with this grammar extension is that the exclamation mark, !, is already used in Java as a logical negation operator.

As asynchronous method calls are typed to future types, the JL5 type checker is also extended to support this. In addition, several checks are implemented that are required by JCoBox, for example, the correct usage of the different class kinds, and the type restriction for fields of immutable classes and static fields.

The compiler is realized by several passes that analyze and/or modify the AST of a single compilation unit (a source file). The following list gives a short overview of the different passes. The passes are divided into passes that can be done before type-checking and passes that require a fully typed AST.

Before type-checking:

1. Add several package imports and static imports to the import list of the file.
2. Make all static fields as well as fields of immutable classes final.
3. Find asynchronous method calls, whose futures are not needed and mark them to be optimized.

After type-checking:

1. Check the class annotations for correct usage.
2. Check the types of static fields and fields of immutable classes.
3. Ensure that standard classes inherit transitively from CoObjectClass.
4. Rewrite field accesses to use wrapper methods.
5. Create wrapper methods for field accesses.
6. Optimize synchronous calls that target this or super to directly invoke unwrapped methods.
7. Find usages of the Java compatible syntax and replace them with corresponding AST nodes.
8. Create wrapper methods for methods. This step also creates wrapper methods for the field access wrapper methods.
9. Replace asynchronous method calls with corresponding code.
10. Replace new expressions where needed by static factory methods.
11. Create static factory methods for creating objects.
12. Rewrite the main method.

5.7.2 The JCoBox Bytecode Rewriter

One disadvantage of the compiler approach is that the compiler has to support the full Java language, which means that, whenever the Java language changes, the compiler has to be adapted accordingly. This is often a very tedious task.

Instead of using a compiler that takes Java source files as input, it is also possible to use a bytecode rewriter that takes Java class files as input. We implemented a JCoBox bytecode rewriter (JCoBoxBR) that rewrites Java bytecode. To use the rewriter, the programmer writes standard Java programs, expressing JCoBox by using the Java-encoded syntax (cf. Section 4.2.5). The Java code is then compiled by a standard Java compiler to JVM bytecode, which is then rewritten by JCoBoxBR (see Figure 5.12). The resulting bytecode then behaves exactly like the bytecode that is generated when using the JCoBox compiler. The implementation of JCoBoxBR uses the ASM bytecode rewriting library for reading, analyzing, and writing JVM bytecode [ASM10].
5.8 Performance

A programming model is only useful in practice if it can be implemented efficiently. To evaluate the performance of the JCoBox implementation, we compare JCoBox with two industry-strength languages that run on the JVM and have some kind of actor abstraction: Scala and Clojure\(^2\).

Scala features an actor library, which belongs to the fastest actor implementations on the JVM [KSA09]. Clojure is a Lisp-dialect targeting the JVM and has the concept of agents to allow for the asynchronous execution of functions. We used Scala v2.7.7-final and Clojure-1.0.0 in all of our benchmarks. We also included Scala v2.7.5-final, because it uses a different actor backend.

As each language has different names for similar things, we use in the following the term actor for coboxes in JCoBox, actors in Scala, and agents in Clojure; and we speak of message sending, when talking about asynchronous invocation of methods in JCoBox, message sending in Scala, and asynchronous invocation of functions in Clojure.

5.8.1 Benchmark Programs

As there is currently no standard benchmark for measuring actor-like languages, we measure performance by three micro-benchmarks. The first two are taken from the Computer Language Benchmark Game [The10], the third one is an example used in [SM08].

Thread Ring creates a ring of \(n\) actors. A single message is then send around the ring \(m\) times, resulting in a total of \(n \times m\) sent messages. We consider two different configurations. The first configuration (ringmsgs) sets \(n\) to 1 000 and increases \(m\) from 1 000 up to 10 000. This configuration measures mainly message sending and receiving performance and is similar to the configuration used in [The10]. The second configuration (ringnodes) sets \(m\) to 10 and increases \(n\) from 100 000 up to 1 000 000. This configuration mainly measures the creation and handling of a large amount of actors. Both configurations have no concurrency, no contention, and there always exists only one unreceived message at a time.

The Chameneos example [The10, KPP03], creates \(n\) (first 3 then 10) chameneos, which all try to meet another chameneos at a single mall to complement their colors.

\(^2\)We also planned to include Kilim [SM08] in our performance evaluation, but at the time of this writing Kilim suffered from a data race, which made it impossible to get reliable measurements.
The number of meets \( m \) is increased from 200,000 up to 2,000,000. This benchmark measures performance of a high frequency of messages under high contention, but the message load is relatively low and the number of actors is also very low. There is a small potential for parallel execution.

The BigPingPong (pingpong) example [SM08] creates \( n \) actors, which sends each other actor a single message, so that \( n^2 \) messages are sent and received. This benchmark measures performance under a high load of simultaneous messages, under low contention, with a medium number of actors and with a high potential of parallel execution.

### 5.8.2 Setup

We ran all benchmarks on five different hardware platforms: An Intel Atom N270 1.6GHz CPU with 1GB RAM (Atom). An Intel Core 2 Duo T7400 2.16GHz CPU with 2GB RAM (Core2Duo). An AMD Athlon dual-core 4850e 2.5Ghz CPU with 4GB RAM (Athlon). An AMD machine with two dual-core AMD Opteron 270 2GHz CPUs and 4GB RAM (Opteron). An Intel Xeon X3220 quad-core CPU with 4GB RAM (Xeon).

The Atom, Core2Duo and Athlon platforms run a 32bit Linux 2.6.31, the Opteron platform runs in a XEN virtual machine with 64bit Linux 2.6.16-xen, and the Xeon platform runs a 64bit Linux 2.6.25. All benchmarks were executed on the Sun JDK version 1.6.16. On the Core2Duo and the Athlon platform the 32bit Server VM, and on the the Opteron and Xeon platform the 64bit Server VM was used. On the Atom platform we used the 32bit Client VM, which is the default on this platform, as well as the 32bit Server VM. All benchmarks were executed with `java -Xmx1024M`, thus with a maximal memory of 1GB.

We followed the advice of Georges et al. [GBE07] and measured each benchmark by running \( n \) JVM invocations, each invocation executing \( k \) benchmark runs. For each JVM invocation we took the mean of the last 10 of the \( k \) benchmark runs. The time of a benchmark run was measured by using the `java System.nanoTime()` method. We then calculated the mean and the 95% confidence interval of the means of the \( n \) JVM invocations. The size of \( k \) was at most \( 10 + 5 \), but could be less if the JVM reached a steady state earlier. A steady state was assumed if the \( \text{CoV}^3 \) of the last 10 runs was less or equal to 0.02. The number of JVM invocations \( n \) was either 10 or less if the size of the 95% confidence interval fell below 3% of the overall mean earlier.

### 5.8.3 Results

The chart in Figure 5.13 gives an overview of all benchmark runs with maximal input parameters. Each point represents the execution time of a single language-program-platform combination relatively to the corresponding JCoBox execution time. The y-axis can also be read as the speedup of JCoBox compared to the other languages. This charts allows for a comparison of the different languages as well as the different

\(^3\text{CoV} \) is defined as the standard deviation \( s \) divided by the mean \( \bar{x} \), see [GBE07] for details.
platforms. Figure 5.14 exemplarily shows the different benchmark runs on the Xeon platform with increasing input parameters, the corresponding plots for the other platforms can be found in Appendix D.

In all cases, JCoBox is the fastest, but the speedup significantly depends on the execution platform and the benchmark under consideration. The pingpong benchmark shows the largest speedup with respect to Scala v2.7.7 and Clojure. In the other benchmarks JCoBox was between 1.5 to 4 times faster than Clojure and Scala. In the ringnodes benchmark Scala v2.7.5 timed out on 3 platforms and showed a large variance on the Atom Client platform.

5.8.4 Discussion

In all our benchmarks JCoBox outperformed Scala as well as Clojure. The largest speedups have been in the pingpong example, which shows that JCoBox can deal with a high load of messges. The lowest speedup was in the chameneos example, which focuses on high-contention. The ringnodes benchmark shows that coboxes are very cheap, i.e., it is possible to create millions of coboxes. Clojure has been surprisingly fast, even though it is a dynamically typed language. We also noticed
a significant difference between Scala v2.7.5 and v2.7.7, where the latter has been significantly slower in 3 of 4 of our benchmarks, and the former ran out of memory in the ringnodes benchmark on some platforms, a known bug, which has been fixed in v2.7.7.

It is always critical to use micro-benchmarks to compare the performance of different languages. It cannot be concluded from these benchmarks that in practice there will be a significant difference in speed between the different languages, as the dominant factor of an application might not lie in the actor framework. However, we believe that in practice, JCoBox will be at least as fast as the compared languages.

5.9 Discussion and Related Work

The implementation of JCoBox meets all functional requirements that have been stated in Section 5.1. This is achieved by a Java compiler extension as well as a bytecode rewriter. We also implemented extensions like the RMI support and a simple deadlock detection. The performance of JCoBox proved to be better or at least as
5.9 Discussion and Related Work

good as other JVM actor implementations. We have not directly evaluated how
JCoBox profit from multiple cores, but the parallel pingpong benchmark indirectly
showed that JCoBox is at least as good as Scala and Clojure in utilizing multiple cores.
The JCoBox runtime is implemented in a scalable way. The main limiting factor is
the implementation of the thread pool, which currently relies on a standard thread
pool implementation.

5.9.1 Task Cooperation

Two performance aspects have not been measured, namely the scheduling of cooper-
ative tasks and the possible number of simultaneous tasks because both mechanisms
are not present in other JVM implementations. In the current implementation a
suspended task always suspends its executing thread. Thus, the number of suspended
tasks is limited by the possible number of threads. In addition, suspending a thread is
an expensive operation. Instead of suspending the underlying thread, a continuation
framework for the JVM, like Kilim \[SM08\], can be used. A suspended task then
only costs the memory for holding its execution stack. In addition, suspending and
resuming a task would be much faster.

5.9.2 Task Execution

The JCoBox runtime currently uses a very simple thread pool implementation, i.e., a
global thread pool executes all tasks. This global thread pool can become a scalability
bottleneck. In addition, the thread pool does not respect cache affinities, because
tasks of one cobox are arbitrarily spread over different threads. One way to change
this implementation is to implement a thread pool based on the concept of work
stealing \[ABP98, BL99, Lea00, RSB09\]. Such an implementation would consist of
n thread pools, called logical processors, where n is equal to the number of physical
processors. Each cobox would be assigned to a single logical processor, which is
responsible for executing tasks of that cobox. Logical processors can steal coboxes
from other processors if they get out of work. This would distribute the single tasks
queue over all logical processor and in addition would improve cache locality as tasks
belonging to the same cobox are more likely to be executed by the same thread.

5.9.3 Distributed Programming

The cobox model is well-suited for distributed programming, supported by the fact
that languages explicitly designed for distribution have a similar programming model
\[MTS05, BBC06, VCMB+07\]. The current JCoBox implementation only features
a proof-of-concept implementation based on RMI. It can already be used to write
distributed programs in an asynchronous style, where the underlying programming
model is, despite network failures, identical in the local and distributed case.
One of the goals of this thesis is to provide a model and language that can be used in practice. Section 5.8 shows that the performance of JCoBox is competitive. This is an important prerequisite to be usable in practice. In this chapter, JCoBox is applied to practice. We show how JCoBox applications are designed and implemented. Before the applications are discussed, this chapter starts with a short introduction on a way how to describe systems that are realized in the cobox model.

6.1 Describing Systems by CoBox Diagrams

One of the advantages of using the cobox model, compared to typical thread-based applications, is that concurrency is not orthogonal to components. This makes it much simpler to describe the design of concurrent component-based software as it is clear from the component view where concurrency can happen. In a thread-based design, where threads are orthogonal to component boundaries, concurrency is completely hidden in component diagrams, which makes it difficult to understand the behavior of concurrent applications without supplementary documentation.

To describe the architecture of systems written in the cobox model, we use cobox diagrams. They have been already used in the previous chapters, without an explicit explanation. CoBox diagrams reflect the runtime structure of systems, similar to component diagrams of UML 2 [OMG10]. A cobox diagram is in general a conservative approximation of the real runtime structure of a system. They describe the normal execution of a system, not regarding error cases. In addition, the system is described after the initialization phase. A precise, formal definition of the semantics of cobox diagrams is left for future work.
6.1.1 Basic Diagrams

A cobox diagram is a graph, where the nodes represent possible objects at runtime, and the edges represent the reference relation. CoBoxes are indicated by rounded boxes that surround the objects that they own. For example, the cobox diagram in Figure 6.1 specifies a system that consists of two coboxes at runtime. One cobox consists of two objects of type A and B, respectively, the other cobox consists only of a single object of type C. There exist three references, namely one from A to B, one from B to C, and one from C to B. As the diagram is a conservative view to the system, some of these references of objects may not exist in every state of the system. We often only show the key objects of a system in a cobox diagram. Many cobox-local helper objects are often left out in the diagrams for a concise presentation. Thus the internal structure of a cobox might be more complex than shown, however, service objects are in general not left out, so that the reference structure between coboxes is always as specified.

![Figure 6.1: CoBox diagram with two coboxes and three objects.](image)

6.1.2 Multiplicities

Object-Oriented systems can in general be of arbitrary size. To be able to still describe such systems in a finite diagram, multiplicities are used. Nodes in a cobox diagram can have multiplicities. They indicate that multiple copies of the same kind of node can exist at runtime. The same kind means that they have the same internal structure as well as the same external referencing. The multiplicity can also be * for arbitrary many nodes.

![Figure 6.2: Mapping of a cobox diagram to a cobox diagram with multiplicities.](image)
6.1.3 Node Groups

Often, object structures are hierarchically structured. To represent this in cobox diagrams, nodes can be grouped into node groups. Node groups allow cobox diagrams to be more precise with respect to multiplicities and the referencing structure. Consider the following example. Suppose there is a cobox $a$ that creates at runtime a helper cobox $b$ to execute some tasks in parallel. Now assume that there are several $a$ coboxes in the system. Without grouping, the cobox diagram would look as follows:

![Cobox Diagram](image)

As the diagram has to be an over-approximation, the information that each $a$ cobox uses exactly one $b$ cobox is lost. In addition, it is not clear anymore that each $a$ cobox only references its helper cobox and not any other $b$ cobox. Using node groups, this information can be expressed as follows:

![Node Grouped Cobox Diagram](image)

6.2 Writing Concurrent Desktop Applications

In this section we show how the cobox model and JCoBox can be used to develop and implement a typical desktop application. Object-oriented desktop applications are in general built by the Model–View–Controller (MVC) pattern \[GHJV95\]. The pattern divides the application into at least three components, namely a model, a view, and a controller component. In a sequential program these components are tightly coupled because they communicate by synchronous method calls. As the view is forced to wait for the other components, an application can quickly become unresponsive if one of its components cannot return from a method call in a short amount of time. If I/O is involved in one of the tasks, the response time can, in general, not be predicted. Another problem of synchronous communication are callbacks \[MTS05\]. For example, if the model notifies its views about a state change and one view calls back to the model, the state of the model might arbitrarily change during the notification process.
6.2.1 CoMusic: Requirements

As a concrete example, we use a very simple imaginary music management application called CoMusic. Figure 6.3 shows a screen shot of how it could look like. The application should manage a list of songs. Songs can be bought from a shop, and bought songs can be played. It should be possible to download multiple songs, to update the list of available songs, and to play a bought song. All these tasks should run concurrently. At all times the application should be able to react to user input.

CoMusic has to be implemented in a concurrent way as the process of downloading a song should not prevent another song from being played, for example. In the typical multi-threaded model of object-oriented languages like Java, these independent control flows would be modeled as threads. As these threads may potentially interact with any component, these components have to be written in a thread-safe way to avoid data races. As communication is in general synchronous in the standard thread-model, deadlocks must be avoided, which is difficult in typical MVC-applications because of the circular dependencies of the participating components. Finally, as threads are conceptually different to components, it is difficult to understand such an application, as active threads are intermixed with passive components.

6.2.2 CoBox Design

A JCoBox program is realized as a set of interacting, parallel running coboxes. In the design phase of JCoBox programs, it is thus decided which coboxes should exist at runtime. Figure 6.4 shows how the CoMusic example can be divided into coboxes.

The model is realized by the SongModel cobox. It holds the information about all songs as well as their download status. To keep the GUI responsive all the time, all GUI-related code is realized as a separate SongGUI cobox, which consists of two parts: a controller object of type SongGUI, which interacts with other coboxes, and a set
of Swing objects, which realize the graphical user interface. As the Swing objects are implemented by standard Java classes and not by JCoBox classes, they can only interact with objects of the same cobox. Finally, download requests are handled by the DownCtrl cobox. It uses additional DownProcess coboxes to handle separate downloads concurrently. Songs are played by the SongPlayer cobox. Internally, it uses a legacy library for playing songs.

Songs are represented by immutable Song objects. This allows all components to efficiently access and share Song objects. In addition, SongStatus objects are used to represent the download status of songs. These objects are realized as mutable transfer objects and are thus copied when passed between coboxes.

From the runtime view of the CoMusic application it is immediately clear which parts of the application can run in parallel. Each cobox represents a potential parallel execution. Inside a cobox there may only be cooperative multi-tasking. In a standard thread-based implementation it would be much more difficult to describe concurrency as threads and components are orthogonal.

### 6.2.3 JCoBox Implementation

After having defined how the architecture of CoMusic should look like, we now show its implementation. Each cobox appearing at runtime is created by instantiating a corresponding cobox class.
The Main CoBox

Every JCoBox program starts by executing the standard `main` method in the initial `Main` cobox (not shown in the cobox diagram). In the `CoMusic` example, the `main` method creates and wires the required coboxes (Listing 6.1). A JCoBox program normally terminates if there are no tasks in any cobox anymore. As in Swing applications it might be the case that there are no tasks at all because the application waits for user input, the default behavior has to be disabled by invoking `setAutomaticShutdown(false)`. Shutting down the application must then be done manually by invoking `shutdown()`.

```java
class CoMusic {
   public static void main(String[] args) {
      setAutomaticShutdown(false);
      SongGUI gui = new SongGUI();
      SongModel model = new SongModel();
      DownCtrl dl = new DownCtrl(model);
      SongPlayer player = new SongPlayer(model);
      gui.init();
      gui.setDownloadHandler(dl);
      gui.setPlayer(player);
      model.addListener(gui);  
   }
}
```

Listing 6.1: The main method of `CoMusic`

Data

The main data of the music application are `Song` objects. To allow all components to efficiently access and transfer these objects, they are made immutable. The implementation of the `Song` class is shown in Listing 6.2.

Note that all fields of `@Immutable` classes are implicitly `final`.

The other data objects are `SongStatus` objects, which hold the current download status of a song and the current play time, when the song is played. As the download status changes over time, the class is defined as a transfer class (see Listing 6.3). `SongStatus` objects are thus copied when they are passed to other coboxes.

The Model CoBox

The model cobox is implemented by the `SongModel` cobox class (Listing 6.4). Its state consists of a map, which maps song IDs to corresponding `SongStatus` objects.
In addition, it has a list of SongModelListener objects, which are notified about state changes. Listeners are notified asynchronously. This has a couple of advantages compared to a synchronous notification [MTS05]. The first advantage is that during the notification process no listener can callback the model, and in particular the list of listeners cannot change. The second advantage is that the model has not to wait until each listener has handled the notification, which allows the model to be quickly available again. Finally, the model is unaffected by any exceptions thrown in listeners. Noteworthy is the fact that the SongModel implements the push-based subject-observer pattern [GHJV95], i.e., the changed data is directly passed to the observers to avoid that observers have to callback to the model to obtain the new data. We made the experience that the push-based subject-observer pattern is much better suited for concurrent, loosely-coupled systems as it simplifies the communication structure and increases parallelism. Note also that the SongStatus objects are transfer objects and are copied when passed to the listeners.

The GUI Component

The graphical user interface is implemented by the SongGUI class (Listing 6.5). It acts as a controller class, which handles all communication to and from Swing objects that realize the GUI. This communication happens in a standard sequential event-based style. By annotating the SongGUI class with @Swing, JCoBox ensures that all code of that cobox is executed by the event-dispatching thread of Swing (cf. Section 4.2.3). It is thus always safe to interact with Swing objects inside such a cobox. But it also means that all code executions inside such coboxes block the GUI. Such blocking,
@CoBox class SongModel {
    private Map<Integer, SongStatus> songs = ... 
    private List<SongModelListener> listeners = ... 
    ... 
    public void updProgress(Song s, int p) {
        SongStatus stat = songs.get(s.ID);
        stat.setProgress(p);
        for (SongModelListener lis : listeners) {
            lis.statusChanged(stat);
        }
    } 
}

Listing 6.4: The SongModel cobox class

however, can be kept to a minimum by delegating requests to other coboxes as soon as possible. For example, the action handling code for the buy button first gets the selected song from the SongTable and then immediately delegates the buy request to the DownCtrl cobox using an asynchronous method call. The SongTable class is annotated with @PlainJava as it inherits from a standard Java class. The SongGUI class communicates in a standard synchronous way with SongTable as the SongTable object lives in the same cobox as the SongGUI object.

The Download Component

As the downloading of songs and song information requires I/O operations, it is put into a separate cobox. The DownCtrl cobox is responsible for managing different downloads. Each single download is then handled by additional DownProcess coboxes (Listing 6.6). The DownProcess has a long running task that downloads a single song. As it should be possible to cancel the download process, the task constantly calls yield() to allow the execution of possible calls to the cancel() method. As the scheduling of tasks is always fair in JCoBox, it is guaranteed that all other tasks are executed before a yielded task is executed again.

The Song Player Component

Songs are played by the SongPlayer cobox. To play songs, a legacy library is used. The library provides a class BasicPlayer, which is shown in Listing 6.7. It has a method open(File) to open a music file. After a file has been opened, one can play the file by calling the play() method. Important to note is that invoking play() uses the calling thread to play the song and only returns to the caller when the song is finished or stopped. Stopping a song can be done by calling the stop() method, which obviously has to be done by a different thread. Finally, it is possible to register a BasicPlayerListener to retrieve events from the player, like the current progress, for example.

To interact with the BasicPlayer the SongPlayer component uses an additional
6.2 Writing Concurrent Desktop Applications

```java
@CoBox @Swing class SongGUI
    implements SongModelListener {
    private DownCtrl dlctrl;
    private SongTable table = new SongTable();
    private JButton buyBtn;

    public void init() {
        ... 
        buyBtn = new JButton("Buy");
        buyBtn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent a) {
                Song s = table.getSelectedSong()
                dlctrl.buy(s); } });
        ... 
    }
    public void statusChanged(SongStatus s) {
        table.updateStatus(s);
    }
    ... // further code omitted

    @PlainJava
    class SongTable extends AbstractTableModel {
        ... // implementation omitted
    }
```

Listing 6.5: The SongGUI class and the SongTable class

helper cobox class PlayTask, for which we show the crucial parts in Listing 6.8. The import part is the implementation of the play() method. As invoking play() on the BasicPlayer blocks the invoking thread, the whole cobox would be blocked for any other activity. We thus have to release the cobox before doing the invocation by using the releaseCoBox method (cf. Section 4.2.3). The PlayTask also acts as listener for the player and implements the progress method. Even though the call to progress is done by legacy Java code, it is thread-safe as calls from legacy Java code are handled equivalent to synchronous calls in JCoBox. As the cobox is not blocked when playing a song, calls to the stop() method can be handled to stop the player.

### Runtime Behavior

To illustrate a typical message flow in the CoMusic application, we show the process of buying a song in a sequence diagram (Figure 6.5). To make clear, which activities run in the same cobox, we draw a cobox boundary around them. Activities, which run in different coboxes, can run concurrently, activities in a single cobox are executed sequentially. An important difference to UML sequence diagrams [OMG10] is that
@CoBox class DownProcess {
  DownCtrl ctrl; SongModel model; boolean canceled; boolean finished;

  // [...] initialization code

  void cancel() { canceled = true; }

  void start(Song s) {
    // [...] open connection and create input and output streams
    byte[] buffer = new byte[4096];
    startBlocking(); // dealing with blocking I/O
    while (!canceled) {
      int nbytes = inputStream.read(buffer);
      if (nbytes == -1) {
        finished = true;
        model! updProgress(song,100);
        break;
      } else {
        outputStream.write(buffer,0,nbytes);
      }
      int progress = // [...] calculate progress
      model! updProgress(song,progress);
      yield(); // process potential cancel calls
    }
    endBlocking();
    ctrl! finished(s);
  }
}

Listing 6.6: The DownProcess class

class BasicPlayer {
  void open(File file) { ... }
  void play() { ... }
  void stop() { ... }
  void addBasicPlayerListener(BasicPlayerListener b) { ... }
  ... // further methods omitted
}

Listing 6.7: The BasicPlayer class.
@CoBox class PlayTask implements BasicPlayerListener {
    final BasicPlayer player;
    final SongStatus songStatus;

    PlayTask(SongStatus song) { ... }

    void play() {
        releaseCoBox(); // release cobox as play() call blocks
        try {
            player.play(); // blocks until stop() is called
        } finally {
            acquireCoBox(); // regain access to cobox
        }
    }

    void stop() { player.stop(); }

    // BasicPlayerListener method
    public void progress(int arg0, long musecs, byte[] arg2, Map arg3) {
        songModel ! setSongPlayMillis(songStatus.getSong(), musecs / 1000);
    }
}

Listing 6.8: The PlayTask class.

asynchronous messages are ordered and may not overtake each other if their source
and target cobox are the same. For example, the two statusChanged() calls from
the SongModel to the SongGUI are guaranteed to be executed in the order in which
they have been sent. This is important as otherwise an outdated status may be shown.

6.2.4 Discussion

Implementing the CoMusic application in JCoBox has been straightforward. The
general structure of the application is similar to a typical sequential implementation,
but components become loosely-coupled and run concurrently. This shows that the
cobox model naturally fits into object-oriented programming. Interacting with legacy
Java code is necessary in practice, but JCoBox offers mechanisms to do this in a
thread-safe way.

The chosen design for the CoMusic application is only one possibility. If the
application has already existed in a purely sequential implementation, one might have
started with a single Swing cobox, which runs all existing code. This implementation
would then be semantically identical to the purely sequential one. Concurrency can
then be introduced step-by-step by introducing new coboxes. In contrast to adding
6.3 The FourWins Game

This section presents the design and implementation of the Connect Four game. It is called FourWins\(^1\). The case study originated in the Software Technology Group of the University of Kaiserslautern as a small concurrent application with a complex behavior. It is in particular difficult to provide a simple implementation using the standard Java thread model. Figure 6.6 shows a snapshot of the FourWins game.

\(^1\)After the german name “Vier Gewinnt”
6.3 The FourWins Game

We first give the original requirements of the game and then add some additional requirements, which were not originally formulated.

![Figure 6.6: Snapshot of the FourWins application.](image)

### 6.3.1 Requirements

The original requirements of the FourWins game are as follows.

- The game is played like the standard connect four game, i.e., there are two players, which have different colors. In the following we assume a *red* and a *blue* player. The blue player always begins. Each player makes a single move. The board of the game consists of 7 columns and 6 rows. On each turn a player can put a single token in one column if the column is not full. A player wins if there are 4 tokens of his or her color in a line, which can be horizontal, vertical, or diagonal.

- Each player should be playable either by a human, via a graphical user interface, or by the computer.

- At all time it should be possible to change sides, i.e., a player should be able to continue to play with the tokens of the other player.
• It should be possible to have multiple observers, which can observe the game, but who cannot play.

In addition we add the following requirements:

• The game must always be reactive, i.e., it should never hang noticibly.
• The user should always be able to stop or quit the game without having to wait more than half a second.
• The algorithm of the computer player should utilize multiple cores to get a nototible speed-up.

The FourWins application is in some aspects similar to the CoMusic application of Section 6.2 because it is a desktop application and has a GUI. However, FourWins does neither need access to the Internet, nor does it require other I/O access, except for interaction with the user via the GUI. The interesting part of the FourWins application is the combination of a computer player and a human player. The players interact in completely different ways with the application. The human uses the graphical user interface, whereas the computer can directly interact with the application logic. This combination gives this example its complexity. An additional interesting point is that the algorithm of the computer player can be written in a parallel way to make use of multiple cores.

6.3.2 Design and Implementation

This section describes design and implementation of the FourWins application in JCoBox. A quite detailed cobox diagram of FourWins is shown in Figure 6.7. The architecture is based on the MVC pattern [GHJV95]. As the interesting aspects concerning the MVC pattern are already covered by the description of the CoMusic example, we do not treat these aspects in detail here. The model is represented by the Board cobox. It holds the state of the current board position. In addition, listeners can be registered, which are updated about state changes. The GameGUI represents the view. It listens for state changes of the Board and is updated accordingly. The GameGUI uses a Swing cobox created by the SwingGUI cobox class. The Swing cobox is responsible for handling all Swing-related code. It manages the event-listener code that is triggered when the user interacts with the GUI and updates the Swing components that represent the board view. In addition, several instances of the GameGUI can exist, a feature which is used to realize game observers.

The Controller

The Controller cobox controls the whole gaming process. It asks the current player to calculate the next move and applies the move to the Board. The controller also handles player change requests and is responsible for starting and stopping the game. In this section we only investigate the communication with the different
players as this is the most interesting part. One design goal is to treat computer
and human player identically from the perspective of the controller. A player must
implement the Player interface, which has a method nextMove() to calculate the
next move as well as a method abortMove() to stop a running calculation. Listing 6.9
shows the relevant parts of the Controller class that interact with the players.
Independently of whether the player is a human or computer player, the controller
sends an asynchronous message to get the next move and cooperatively waits for
the result. It then applies the returned move to the board and finally switches to the
next player. In addition, it is checked whether there have been any requests to switch
players. As the future claiming is done cooperatively, the swapSides method can be
handled, while the controller is waiting for the results.

The Computer Player

The computer player is realized by the ComputerPlayer cobox class. Furthermore,
several Worker coboxes are used to calculate the next move. The calculation can then
be done parallel and profit from multiple processors. Listing 6.10 shows a simplified
version of the implementation of the ComputerPlayer class. The basic idea is to
create a worker for every column. The computer player then asynchronously invokes
a method on each worker to calculate the winning chance when doing a move with
Chapter 6 Practical Evaluation

```java
@CoBox class GameController {
    Board board;
    boolean stopped;
    boolean swapSidesRequest;
    ... // further fields

    private void gameLoop() {
        while (!stopped) {
            Player currentPlayer = getCurrentPlayer();
            Move move = currentPlayer!nextMove().await();
            if (handleSwapRequest())
                continue;
            board!applyMove(move, currentPlayer).await();
            if (handleSwapRequest())
                continue;
            nextPlayer();
        }
    }

    private boolean handleSwapRequest() {
        if (!swapSideRequests)
            return false;
        ... // swap players
    }

    public void swapSides() {
        swapSidesRequest = true;
        getCurrentPlayer()!abortMove();
    }
    ... // further methods
}
```

Listing 6.9: Simplified implementation of the Controller class.

this column. All workers do their calculations in parallel. After the computer player has sent the messages, it collects all results by claiming the futures of the calls. It then searches for the best result and returns it. In the actual implementation, a worker can have further subworkers, so that the number of workers can be larger than the number of columns. The presented algorithm has the disadvantage that many board configurations are calculated several times by different workers. To prevent that a board configuration is calculated twice, all workers share a common MoveCache object to share calculated configurations. However, as each worker now has to query the move cache before it calculates the value of a board configuration, the move cache becomes a critical scalability bottleneck. Whereas we first implemented it by a single cobox, we later exchanged that implementation by an implementations based on parallel data-structures from the java.util.concurrent package, for
maximal performance. This showed that the cobox model has its limitations when it comes to applications that require parallel access to a central data component. However, JCoBox allows for using standard thread-safe objects, so it is possible to exchange such critical components by optimized implementations. It is future work to investigate how to extend the cobox model to allow for coboxes that allow for internal parallelism in a safe way. The parallel implementation of the computer player showed in practice a speed up of about 3 on a machine with 4 cores compared to a sequential implementation.

The Human Player

As already mentioned above, the complexity of the FourWins example comes from the fact that human and computer player are implemented in a completely different way. Still, from the perspective of the controller, both components should have the same interface. The implementation of the HumanPlayer is given in Listing 6.11. When
Listing 6.11: Simplified implementation of the HumanPlayer class.

the nextMove() method is called the component cannot directly return the result. Instead the game user has to click a certain button on the GUI. The HumanPlayer thus registers itself at the GUI for getting a notifyMove call by the GUI when the user clicked a move button (Line 4). After that, the move buttons of the GUI are enabled by sending the enableMove() message (Line 5). The HumanPlayer must now wait for notifyMove calls. For this a new promise is created (Line 6), which is resolved by the notifyMove method (Line 11). Finally, it waits for a future of that promise and returns its value (Line 7). This example shows how promises can be used as a communication and synchronization mechanism between tasks.

6.3.3 Discussion

The resulting JCoBox implementation of FourWins fulfills all requirements stated above. The application could be realized by several loosely coupled components that interact by asynchronous method calls. Computer player and human player could both be implemented by using the same interface and the game controller interacts with each of them in the same way. Like in the CoMusic example, all Swing related code is realized by using a Swing cobox. This allows the GUI to stay responsive all the time. Even when the computer player calculates its move it is always possible to interact with the GUI and stop the game, for example. Not shown in this section is that the computer player can also run distributed on a different machine. This was realized without changing the architecture of the system, due to the already distributed structure of the application. During the implementation we found that we required to use a parallel data-structure from the JDK to achieve acceptable performance. Due to the mutual exclusion of tasks in a cobox such parallel data-structures are not possible to implement efficiently in JCoBox. As JCoBox code can interact with thread-safe legacy Java objects, however, it is possible to use specialized
6.4 The CoCoME Example

The two exemplary applications in this chapter so far show how to design and implement typical desktop applications in JCoBox. In this chapter, the cobox model is applied to a distributed information processing system. As an example serves a point of sale (POS) system [Lar04] also know as the Common Component Modeling Example (CoCoME) example [RRPM08].

The CoCoME is a system that manages sales, which occur at supermarkets, for example. It covers the whole sales process, from the cash desk where the cashier enters products, to a central database that stores product information and gives a stock overview. The full CoCoME can manage multiple stores and has an enterprise server to centrally manage the system. In this section, we are only regarding a simplified version of the CoCoME, namely one with only a single store.

6.4.1 Informal Description

A store consists of a store server, where the current product stock and the product information is hold in a database, called the store inventory. Each store has a store client, which is connected to the store server and provides a graphical user interface, which can be used by the store manager to view and modify the product database. A store can have one or more cash desks, which are organized in a so-called cash desk line. The cash desks are the places where the customers pay their purchases. A cash desk consists of the following components:

- A cash desk PC, which runs the cash desk software.
- A cash box, which is used by the cashier to control the payment process. It has a special purpose keyboard to enter prices and product identifiers.
- Several peripherals like a barcode scanner, a card reader for electronic payment, a printer to print receipts, and a display to show customer-relevant information.

The cash box and all other peripherals are connected to the cash desk PC. Further requirements and more details can be found in the CoCoME document [RRPM08].

6.4.2 Design and Implementation

The cobox design of the CoCoME example is shown in Figure 6.8. It is a simplified version of the implementation of Worret [Wor09a], which implements the CoCoME as a product line. It is straightforwardly derived from the informal description. As all mentioned components run concurrently and are loosely coupled, each of them is realized as its own cobox. A real implementation of the CoCoME would consist of

libraries when needed.
concrete hardware for each of the peripherals. In our implementation we simulate the hardware by Swing-based GUIs, which are not shown in the diagram. The application is written with RMI support, which allows the store server and the cashdesks to run on different machines, for example. Otherwise the implementation did not reveal aspects which are not already covered in the previous sections, so we do not go into further details in this section.

6.4.3 Discussion

The CoCoME could be designed and implemented in a straightforward way with the cobox model. This shows that information processing systems are an ideal domain for the cobox model. As these systems are already distributed, the architecture can be directly expressed by using coboxes. In addition, asynchronous communication between components is also already given due to their distribution.

6.5 Experience and Discussion

Our experience with JCoBox has been very promising. All example applications could be successfully implemented with JCoBox. The design of these applications in general resemble that of standard object-oriented design. This comes due to
the fact that the cobox model uses object-oriented components as the unit of state and behavior, a concept which seamlessly matches object-oriented thinking. Using the cobox model does not require a radical new way of thinking. As typically the communication between components is only a very little part of the implementation of a program, most parts of an application, namely the component-internal parts, are written in a standard sequential way. As the sequential part of the JCoBox semantics is equivalent to standard sequential Java, programmers that are used to Java can immediately write JCoBox programs without having to learn a complete new programming model. In addition, when writing component-internal code, the programmer is completely relieved from concerning concurrency as tasks of different coboxes run isolated to each other. There are mainly two things which may be a difficulty for new programmers. First, to become used to the fact that objects belong to certain components and that it is important to which component an object belongs, and second to become used to asynchronous method calls. In practice, most objects are near objects and it is clear which objects can be far objects. Nevertheless, it means that a programmer must mentally assign objects to their corresponding coboxes, which requires some learning effort. Asynchronous method calls are a concept which, we believe, is a bit more difficult to fully exploit. It is in the beginning difficult to give up the synchronous communication with its total certainty. Asynchronous communication require the programmer to accept that something eventually will happen, but maybe not at once. The full concurrency potential of an application can only be achieved when learning to effectively use asynchronous communication as every synchronous call means that the calling task has to stop running. One aspect of asynchronous method calls, which we experienced, is that a partial ordering is essential to be able to use the full potential of asynchronous communication. As it would be cumbersome to implement an ordering on top of unordered messages by hand, a programmer instead waits overly conservatively for the result of asynchronous method calls to ensure a correct ordering of subsequent calls. In fact, in an early version of JCoBox, method calls were unordered. After gaining some experience, we changed that decision in favor of ordered method calls.

The structure of an application is in general similar to a standard sequential implementation, but components become loosely-coupled and run concurrently. It is, however, sometimes advantageous to restructure an application. The ubiquitous subject-observer pattern in desktop applications is often better implemented in a push-based approach, instead of a pull-based approach [GHJV95, p. 298]. In the push-based approach the subject notifies its observers not only about a state changed, but directly passes the modifications to the observers. Thus, observers are not required to call back to the subject to obtain the updated state. Such a design is often simpler, even in sequential programs [MTS05]. However, in concurrent systems, the push-based variant has the additional advantage that parallelism can be better exploited due to potential pipelining. Furthermore, in the distributed settings, additional round trips are avoided.

JCoBox takes much pain away from concurrency. As JCoBox prevents data races
by design and makes it much more difficult to create non-deterministic deadlocks, concurrency-related bugs appear only rarely. Deadlocks appeared during the development of the example applications, but in all cases they were deterministic and the source of the bug could be found very quickly. Data inconsistencies and high-level data races can appear in JCoBox when a task yields control and afterward relies on assumptions made before the yield. We experienced such inconsistencies, but found the bugs in general very quickly, because of the restricted set of possible interleavings in the cooperative task setting. The issue that cooperative multitasking requires that all tasks cooperate to ensure fairness was never a problem. This comes from the simple fact that tasks are local to a component and thus the developer of the component has full control over all possible tasks. One argument that showed that JCoBox is easy to learn is that the CoCoME example was written by a graduate student after a short introduction into the language. He managed to write a concurrent and distributed application without experiencing much problems with concurrency.
CHAPTER 7

Conclusion

Concurrency is getting into desktop computers and thus into mainstream programming. Object-oriented programming is the dominant programming model for this application domain. The concurrency model of most OOLs, like Java and C#, is based on preemptive threads working on a shared state. This programming model is unsafe by default, requiring the programmer to prevent data races by using locks or monitors. Programs written in this model are difficult to understand and to test, and are not very modular. In addition, the ubiquitous synchronous communication makes it difficult to prevent deadlocks. An alternative concurrency model for OOP is the active object model. Instead of expressing concurrency by threads, which are orthogonal to objects, the unit of concurrency are the objects themselves, leading to a more natural way to express concurrency in OOLs. Active objects communicate by asynchronous method calls instead of synchronous method calls. This decouples the concurrent entities, leading to loosely coupled systems, which are easier to understand and test. This is because active objects can be understood in isolation, without regarding the concurrency of their environment. The remainder of this chapter summarizes and discusses the contributions of this thesis and gives an outlook to future work.

7.1 Contributions

The state and behavior of objects is often constituted by interacting groups of objects. These groups form conceptional runtime components with a runtime interface that can consist of multiple objects. This thesis presents a programming model that generalizes the idea of active objects to concurrently running, isolated, runtime components, called coboxes. CoBoxes communicate by asynchronous method calls with standard objects as targets. Asynchronous method calls are a type-safe communication mechanism, which is compatible with standard class-based object-orientation. The state of a cobox consists of multiple objects, where some of them can act as service objects. The behavior of a cobox is realized by a set of cooperatively
scheduled tasks. Multiple cooperative tasks allow the combination of reactive and active behavior in a safe way and allow for waiting for conditions without blocking a cobox for other activities. Synchronization of tasks is based on the concept of promises and futures leading to a data-driven synchronization. Futures also allow for realizing synchronous communication on top of asynchronous method calls. Futures can be claimed exclusively or cooperatively allowing for flexible controlling of the cobox state while waiting for futures. Data can be transferred between coboxes by using transfer objects, which are passed-by-copy, or immutable objects that are passed-by-reference. This thesis makes the following contributions:

- It develops and describes the cobox model—a novel concurrency model for object-oriented programming, which generalize the active object model to concurrent runtime components.

- The cobox model is formalized in a core calculus, which we prove sound and show that its sequential core is equivalent to sequential Java.

- The core calculus is implemented in the rewriting logic framework Maude.

- The cobox model is realized as a programming language, which extends sequential Java with a minimal set of additional constructs to express the cobox model.

- The language is implemented by a compiler that generates Java code and a bytecode rewriter that rewrites JVM bytecode and the performance of the implementation is shown to be competitive to state-of-the-art JVM actor implementations.

- Finally, the programming model and language is evaluated by designing and implementing several concurrent and distributed object-oriented applications.

### 7.2 Discussion

It is always difficult to evaluate a new programming model and language, especially if it introduces new concepts, which have not existed before. There are several criteria, which cannot be objectively evaluated easily, like whether a programming model is easy to learn and to use, for example. In addition, the acceptance of programming languages in industry mainly depends on good documentation and tool support. There are several aspects of the cobox model and JCoBox, however, which makes us believe that concurrency in object-oriented programming languages is much better addressed with our approach than with the standard thread-based model. The most important point is that coboxes are a modular approach to handle concurrency. The main weakness of the thread-based model is that it is not amenable to standard unit-testing due to the overwhelming number of possible execution traces.
In addition, locks have global effects and components that use locks can, in general, not be completely tested in isolation. CoBoxes on the other hand can be tested in isolation. The behavior of coboxes is independent of the state of its environment as coboxes can only communicate by using asynchronous method calls. In addition, as the behavior of a cobox is realized by cooperatively scheduled tasks, the number of task interleavings, and thus execution traces, are greatly reduced, making coboxes amenable to standard unit-testing.

This thesis shows that the cobox model can be integrated into an existing object-oriented language with only a minimal set of additional constructs and a competitive performance. The language extension mainly consists of two new concepts: co-box classes and asynchronous method calls. Both mechanisms naturally fit into object-oriented programming. We have successfully implemented several concurrent applications of different kinds in JCoBox. This makes us confident that programmers that are used to standard OOLs can quickly learn and use the extended language. As standard sequential programming is preserved in the cobox model, most code can be written without taking concurrency into account. This is in contrast to multi-threaded programming, where a programmer always has to keep possible thread interleavings and locking disciplines in mind.

The cobox model is a generalization of the active object model. Compared to the standard active object model it has the advantage that it allows for a more complex internal state consisting of multiple objects, a model similar to ASP \[CH05\]. This makes it possible to combine standard sequential programming with active objects. The cobox model additionally supports multiple service objects. This means that the external interface of a cobox can consist of multiple objects, which is important in OOP to be able to support complex runtime interfaces. CoBoxes can be considered as containers for objects. This concept is similar to the programming model of the E programming language \[MTS05\]. Standard active objects are typically single-threaded, i.e., only one thread can exist in an active object. This makes it difficult to realize multiple independent control flows, which is important for combining active and reactive behavior. The cobox model adopts the Creol approach \[JOY06\] of realizing the internal behavior, using cooperatively scheduled tasks, which are tightly integrated with the concept of futures and promises to allow for a data-driven synchronization. All in all, the cobox model does not introduce radically new ideas. Instead, it combines several approaches into a concise and generalized model and provides a language implementation on top of Java with a competitive performance.

### 7.3 Outlook

This thesis proposes a novel programming model for concurrent and distributed object-oriented programming and demonstrates that the cobox model and its language realization are very useful for designing and implementing concurrent and distributed, object-oriented software. However, this can only be considered as a first step towards
mastering concurrency in OOP. There are several ways to further extend and exploit the cobox model and the JCoBox language. This section provides a brief outlook on further possibilities.

Parallel State Access

Sharing of state between coboxes is currently limited to immutable objects. All other state is encapsulated in coboxes. As tasks of coboxes run mutually exclusive, parallel access to mutable state is not possible in the cobox model. This means that parallel access to mutable state is not possible in the cobox model. For example, highly parallel data-structures are not possible to implement in the cobox model. In addition, it is impossible to efficiently implement certain data-parallel algorithms. For example, a parallel in-place sorting of an array is not possible to implement in JCoBox. Also algorithms, which allow data races for efficiency [Boe05] cannot be realized in the cobox model.

We argue that the cobox model should be used as a high-level programming model, where it is possible in certain cases to “escape” from the high-level model and use special purpose libraries that address these issues [LSB09, Lea00]. In addition, parallel access to shared state is possible if the state is immutable. For example, finding the maximum value in an immutable array can be implemented in a parallel way in the cobox model. Future work should investigate how parallel access to state can be addressed in the cobox model in a safe way.

Different Local Execution Models

Currently, the cobox model defines a fix scheduling scheme for cobox-local tasks, namely deterministic, cooperative multi-tasking. Sometimes it is desirable to have a more flexible scheduling, maybe even a parametric scheduling of tasks. The cobox model can be extended to support other local execution models as long as they do not change the external communication mechanisms of coboxes. For purposes of modeling, for example, a non-deterministic scheduling like in Creol [JOY06] could be applied. It would also be possible to allow tasks to run truly parallel in a cobox. The behavior of such coboxes would be similar to coboxes where each task yields after each instruction with non-deterministic cooperative scheduling. From the perspective of the environment, the local execution model is irrelevant. As communication is only possible via asynchronous method calls, a caller cobox cannot determine the local execution model of the target cobox. This allows for combining coboxes with arbitrary execution models, embedded in a common concurrency and communication model.

Hierarchical Components

The cobox model, described in this thesis, focuses on primitive coboxes. These coboxes only consist of objects and tasks. In particular, it is not possible to explicitly
compose coboxes to define new coboxes. The composition of coboxes is based on the standard OOP composition mechanisms, namely by reference passing. Such composed coboxes, however, do not form coboxes again, they rather form concurrent subsystems. From a behavioral point of view, such subsystems can be treated as conceptional components again. Similar to the box model \([PHS07]\), it would then be possible to define hierarchical components. The internal behavior of such composite components is then defined by several coboxes. The externally visible behavior, however, cannot be distinguished from a primitive cobox. Hence these components can be, in turn, used to form larger components. Another approach is to make the hierarchy of coboxes explicit in the semantics as done in \([SPH08]\). Parent coboxes can then even have additional control over nested coboxes. Finally, a component model can be defined that is independent of coboxes, but use coboxes as primitive units of composition. This model could either be integrated into the language, like in ArchJava \([ACN02]\) or JCowch \([SRGPH08]\), or can be defined above the language level as done in ASP \([CH05]\) or in coordination languages such as Reo \([Arb04]\), for example.

**Distributed Systems**

The cobox model is well suited for describing distributed systems as coboxes communicate by message-passing. Programming languages, which are explicitly designed for distributed systems, like E \([MTS05]\) and AmbientTalk/2 \([VCMB+07]\), have a programming model, which is similar to the cobox model. However, the cobox model, currently, does not support to explicitly describe distribution. It has no concept of computation nodes to specify distributed locations, for example.

The current implementation supports distributed programming based on RMI. However, the programmer has to explicitly work with RMI mechanisms, i.e., the underlying distribution technology is not hidden from the user. A future goal could be to provide mechanisms that abstract from the underlying technology and to express distribution aspects directly in the model and language.

**Efficient Multi-Core Execution**

Multi-core architectures of the near future will most likely have architectures with a non-uniform memory access (NUMA) model \([CGS+05]\). This imposes additional challenges for an efficient execution of concurrent programs. The cobox model is well prepared for these architectures as coboxes already partition the state space. The current implementation, however, does not exploit the cobox runtime structure to efficiently assign coboxes to processor nodes. On NUMA architectures this can result in bad performance due to far memory accesses when the same cobox is executed by processors with different local memories. A specific scheduler that explicitly uses the knowledge of the cobox semantics can ensure that coboxes are mostly executed by the same processor, which would significantly improve the execution performance on
these systems.

**Specification and Verification**

The cobox model provides a component model for object-oriented programs in a concurrent setting. CoBoxes define a clear runtime boundary, which makes it possible to precisely describe the behavior of coboxes by tracing boundary-crossing messages [PHS07, PHGS08]. One important aspect is that, unlike for components in a standard multi-threaded setting [ÁGS08], messages are independent of threads and do not carry thread identifiers. This greatly simplifies the message behavior of coboxes. In addition, asynchronous communication decouples the sender from the receiver and makes it possible to verify the local behavior of a task without taking the receiver state into account. In particular, reentrant calls need not to be considered. Like in Creol [dBCJ07], state invariants of the cobox must only be established when the task explicitly releases its control. This can only happen when cooperatively waiting for a future, explicitly yielding, or when the task terminates. The cobox model, however, has additional challenges compared to Creol as the state of a cobox can consist of multiple objects and the external interface of a cobox can consist of multiple service objects. Future work has to address these issues.
A.1 Dynamic Semantics

For the dynamic semantics, we provide the precise definition of substitution.

A.1.1 Substitution

Definition A.1 The substitution of all occurrences of the free variable $x$ in expression $e$ with expression $e'$, denoted by $[e'/x]e$, is defined as follows.

$$
[e'/x]x \triangleq e'
$$

$$
[e'/x]y \triangleq y', \quad \text{if } x \neq y'
$$

$$
[e'/x]e \triangleq e, \quad \text{if } e \in \{ \text{null, yield, new } c, \text{promise } \tau \}
$$

$$
[e'/x](e.f) \triangleq ([e'/x]e).f
$$

$$
[e'/x](e.f = e'') \triangleq ([e'/x]e).f = [e'/x]e''
$$

$$
[e'/x](\text{new } c \text{ in } e) \triangleq \text{new } c \text{ in } [e'/x]e
$$

$$
[e'/x](e.n(e)) \triangleq ([e'/x]e).n([e'/x]e)
$$

$$
[e'/x](e!n(e)) \triangleq ([e'/x]e)!n([e'/x]e)
$$

$$
[e'/x](e.get) \triangleq ([e'/x]e).get
$$

$$
[e'/x](e.await) \triangleq ([e'/x]e).await
$$

$$
[e'/x](e.fut) \triangleq ([e'/x]e).fut
$$

$$
[e'/x](e.resolve e'') \triangleq ([e'/x]e).resolve [e'/x]e''
$$

$$
[e'/x](\text{let } x = e \text{ in } e') \triangleq \text{let } x = e \text{ in } e'
$$

$$
[e'/x](\text{let } y = e \text{ in } e'') \triangleq \text{let } y = ([e'/x]e) \text{ in } ([e'/x]e''), \quad \text{if } y \notin \text{FV}(e')
$$
In addition, the substitution is defined on sequences of expressions:

\[
\begin{align*}
[e'/x] \equiv & \bullet \\
[e'/x] (e \cdot \bar{e}) \equiv & ([e'/x] e) \cdot [e'/x] \bar{e}
\end{align*}
\]

A.2 Type Soundness

In this section, additional definitions, lemmas, and proofs are given that regard the type soundness proof.

A.2.1 Properties of Data Transfer

Data is transferred between coboxes by using the \textit{copy} function. As its definition uses the \textit{reach} function (cf. Table 3.10), we first show some properties of that function.

As the definition of \textit{reach} is recursive, it is not immediately clear that it is well-defined. We show that \textit{reach} is well-defined by showing that it is deterministic and that there is no cycle in the its definition, i.e., it always terminates. In addition, we show that \textit{reach} is total.

\textbf{Property A.1} \textit{reach} is total, deterministic, and has no cycles.

\textit{Proof. Totality.} We show that for all $O$ and $\bar{v}$, \textit{reach} has at least one value. Assume $O \neq \emptyset$ and $\bar{v} \neq \bullet$, otherwise either the first or the second definition of \textit{reach} applies. In that case either the third, fourth, or fifth definition of \textit{reach} applies.

\textit{Determinism.} We prove determinism by showing that either the conditions of the definitions of \textit{reach} are either distinct or they result in the same value. Assume $O = \emptyset$ and $\bar{v} = \bullet$. Then either the first definition or the second definition can apply, but both result in the same value $\emptyset$. If $O = \emptyset$ and $\bar{v} \neq \bullet$ only the first definition applies. If $O \neq \emptyset$ and $\bar{v} = \bullet$ only the second definition applies. If $O \neq \emptyset$ and $\bar{v} \neq \bullet$, then one of the last three definitions applies. As the conditions of the last three rules are distinct, only one of them can be applied.

\textit{Termination.} We first define a well-founded ordering relation $<_{\text{reach}}$ on the arguments of \textit{reach}:

\[
O, v <_{\text{reach}} O', v' \iff |O| < |O'| \lor (|O| = |O'| \land |v| < |v'|)
\]

$<_{\text{reach}}$ is well-founded because it has the minimal element $\emptyset, \bullet$. With this ordering relation, it becomes immediately clear that \textit{reach} always terminates, as for each right side of its definition either, (1) no recursion appears (for the first and the second definition), (2) the first argument $O$ of \textit{reach} becomes smaller (for the third and fourth definition), or (3) the size of the first argument stays the same and the second argument becomes smaller (last definition). \hfill \square

This completes the proof of Property A.1.
We now show that the set of objects returned by \textit{reach} only contains transfer objects and unresolved future objects.

\textbf{Property A.2} Assume \textit{reach}(O, \vec{v}) = O'. Then for all \( o \in O' \) either \textit{transferobj}(o) or \textit{futobj}(o) \land v_o = \epsilon.

\textit{Proof.} Immediately from the definition of \textit{reach}, as only objects having the demanded properties are added to the result set of \textit{reach}.

Furthermore we show that the set of objects returned by \textit{reach} is a subset of the argument set, with the restriction that futures in the result set might be unresolved.

\textbf{Property A.3} Assume \textit{reach}(O, \vec{v}) = O'. Then for all \( o \in O' \) either \( o \in O \), or \( o = \text{ref}(t, \kappa_p, \epsilon) \) and \( \text{ref}(t, \kappa_p, v_\epsilon) \in O' \).

\textit{Proof.} Immediately from the definition of \textit{reach}.

We show that the function \textit{reach} results in a transfer-closed set of objects, i.e., all transfer objects and futures referred by fields of transfer objects of the result set \( O' \) are contained in \( O' \) again.

\textbf{Lemma A.1} Assume \textit{reach}(O, \vec{v}) = O'. For all \( \kappa_p, t \in \text{orefs}(O') \cup \vec{v} \), it holds that if \( \text{ref}(t, c, \vec{v'}) \in O \) and \textit{transfercl}(c), then \( \text{ref}(t, c, \vec{v'}) \in O' \), and if \( \text{ref}(t, \kappa_p, v_\epsilon) \in O \) then \( \text{ref}(t, \kappa_p, \epsilon) \in O' \).

\textit{Proof.} By the definition of \textit{reach}, we can see that whenever an object is added to the result set, all its field values are recursively evaluated by \textit{reach}. Whenever a reference appears in the argument values of \textit{reach} that refers to a transfer object these objects are always added to the result set. This ensures that all transfer objects and futures that are referenced by objects of the result set are contained in the result set.

\textbf{A.2.2 Proof of the Well-Formedness Lemma}

This section provides the proof of \textbf{Lemma 3.3}.

\textit{Proof.}

\textbf{Assumptions.} The following assumptions are given.

\[ \text{wf}(K) \quad (A1) \quad K \rightarrow K' \quad (A2) \]
Goals. We have to show that \( K' \) is well-formed, i.e., \( \text{wf}(K') \). This means we get the following goals, which have to be shown.

\[
\forall k \in K' \cdot \text{wf}(k) \land \text{wttransfer}(K', k) \quad (G1) \quad \text{uniqueids}(K') \quad (G2)
\]

\( \text{wf}(K') \quad (G) \)

It should be clear that (G2) is always satisfied because new component identifiers are always fresh. The remaining goal is (G1), which we show by a case analysis on all rules of the operational semantics. For each \( k \in K' \) must be shown that \( \text{wf}(k) \) and \( \text{wttransfer}(K', k) \). Only rules can invalidate the well-formedness condition that either

Case (R-NewObjLocal) (C2). This rule creates a new object. The uniqueness of its identifier is guaranteed by choosing a not-existing one.

Case (R-FutAwait) (C3). This rule adds a new task to the suspend set of a cobox. The new task always has the form \( \tau(e \sqsupset [r, \text{get}]) \), hence (WF-SuspendSet) is satisfied.

Case (R-NewObjFar) (C2) (C4). This rule creates a new object \( o \) in a different cobox. The identifier \( \iota \) is uniquely chosen. The result of the creation is a new far reference \( \kappa'_b \iota \). By the precondition \( \text{plaincl}(c) \), object \( o \) is a normal object. So that (WF-Transfer) remains satisfied.

Case (R-NewCoBox) (C1) (C2) (C4). Similar to the case above, but also creates the target cobox, which, however, does not invalidate any well-formedness condition.

Case (R-PromNew) (C1). This rule creates a new unresolved promise, which satisfies (WF-PromInit).

Case (R-PromResolve) (C2) (C4). This rule resolves an unresolved promise. This introduces new objects \( O' \) and may introduce new references. The uniqueness property of \( O' \) is obtained by the definition of \( \text{copy} \). By Property 3.2, all objects of \( O' \) are either futures or transfer objects. Hence we obtain (WF-PromRes). It remains to show that all new far references refer to normal objects, which is given by Lemma 3.1 and Lemma 3.2.
Case (R-PROMFUT) (C2). This rule only creates a new future with a fresh identifier.

Case (R-FUTRESOLVE) (C2) (C4). This rule resolves a future. It may create new objects due to the copying of transfer objects and may also introduce new far references. The proof is similar to that of rule (R-PROMRESOLVE). In addition it requires Property 3.3 to obtain the uniqueness of object identifiers.

Case (R-ASYNCALLLOCAL) (C1). This rule only creates a new fresh unresolved promise, which is well-formed according to (WF-PROMINIT).

Case (R-ASYNCALLFAR) (C1) (C2) (C4). This is the most complex case. It creates a new promise, which is well-formed according to (WF-PROMINIT). It also copies potential transfer objects and futures from the source cobox to the target cobox. That this does not invalidate well-formedness can be shown as done in the cases for the rules (R-FUTRESOLVE) and (R-PROMRESOLVE).

A.2.3 Auxiliary Lemmas

Before we can prove the main theorem, we first introduce several auxiliary lemmas.

A standard property of many typing relations also hold for the typing relations of JCoBox, namely that each term is uniquely typed by a single type rule and thus the relation can be inverted (cf. [Pie02]). In the following, we will use this lemma without explicitly mentioning it.

Lemma A.2 (Inversion) The typing relations, $\vdash_p$, $\vdash_d$, $\vdash_h$, $\vdash_e$, $\vdash_k$, $\vdash_t$, and $\vdash_o$ can be inverted.

Proof. Immediate from the definition of the typing relations.

Another standard property of JCoBox is the substitution lemma (cf. [Pie02]), which states that substituting a variable of type $\tau$, by a term, which is typed to a subtype of $\tau$, is always type-correct, and the resulting expression is typed to a subtype of the type of the original expression. In our case, substitution is only done with values.

Lemma A.3 (Substitution) If $\Gamma, x : \tau'; \Sigma \vdash_e e : \tau$ and $\Gamma; \Sigma \vdash_e v : < \tau'$ then $\Gamma; \Sigma \vdash_e [v/x]e : < \tau$.

Proof. The proof proceeds by induction on the typing derivations, by using the definition of substitution (Definition A.1, Page 163) and the induction hypothesis.

Another standard property is that extending the variable typing has no effect on the typing of expressions.

Lemma A.4 (Variable Typing Extension) If $\Gamma; \Sigma \vdash_e e : \tau$ and $x \notin \text{dom}(\Gamma)$ then $\Gamma, x : \tau'; \Sigma \vdash_e e : \tau$, for any $\tau'$.
Proof. Only the typing of variables is affected by $\Gamma$. If an expression $e$ can be typed under $\Gamma$ then all free variables in $e$ can be typed by $\Gamma$. Adding the typing of a variable $x$ to $\Gamma$, where $x \notin \text{dom}(\Gamma)$ thus cannot affect the typing of $e$ as $x$ cannot be free in $e$. \qed

Lemma A.5 An extended reference typing does not change typing. Let $\Sigma \subseteq \Sigma'$, then

a) if $\Gamma; \Sigma \vdash e : \tau$, then $\Gamma; \Sigma' \vdash e : \tau$,

b) if $\Sigma \vdash t$, then $\Sigma' \vdash t$,

c) if $\Sigma \vdash o$, then $\Sigma' \vdash o$,

d) if $\Sigma \vdash k$, then $\Sigma' \vdash k$.

Proof. If an expression $e$ can be typed under $\Sigma$ then $\Sigma$ contains all references that appear in $e$. Thus additional references in $\Sigma'$ cannot change the typing of $e$. The same holds for tasks, objects, and configurations. \qed

The next lemma states that a correctly typed evaluation context implies that its subexpression is correctly typed.

Lemma A.6 Suppose $\Theta \vdash e \square[e] : \tau$, then there exists a $\tau'$ with $\Theta \vdash e : \tau'$. \qed

Proof. All expression typing rules require that subexpressions are correctly typed. \qed

Next we show that an expression in an evaluation context can be safely replaced by another expression if the type of that expression is a subtype of the type of the replaced expression.

Lemma A.7 (Typing Context) Assume $\Theta \vdash e : \tau$, $\Theta \vdash e' : < \tau$, and $\Theta \vdash e\square[e] : \tau'$, then $\Theta \vdash e\square[e'] : < \tau'$.

Proof. Proof by induction on the shapes of $e\square$ and a case distinction on the basic shapes. We assume

$$\Theta \vdash e : \tau \quad (1) \quad \Theta \vdash e' : < \tau \quad (2) \quad \Theta \vdash e\square[e] : \tau' \quad (3)$$

Case $e\square = \square$. Hence $e\square[e] = e$ and $e\square[e'] = e'$. By (1), (2), and (T-SUB), we directly obtain $\Theta \vdash e\square[e'] : \tau'$.

Case $e\square = e'.f$. Hence $e\square[e] = e'[e].f$ and $e\square[e'] = e'[e'].f$. By (3) and Lemma A.6, $\Theta \vdash e'[e'] : \tau''$ for some $\tau''$. Applying the induction hypothesis we obtain $\Theta \vdash e'\square[e'] : < \tau''$. As we do not allow field hiding, a field has the same type in all subtypes. Thus $\Theta \vdash e'\square[e'].f : \tau'$.

Case $e\square = e'.f = e''$. Similar to the case above.
Case $e \Box = v.f = e'_\Box$. Similar to the case above.

Case $e \Box = \text{new } c$ in $e'_\Box$. As the type of the extended new expression is independent of the type of the target expression (T-NEWIN), it is always $c$ the case can be concluded immediately.

Case let $e \Box$ in $e''$. Immediately by Lemma A.3.

Case $e \Box = e'_\Box ! m(\overline{e})$. Hence $e \Box [e] = e'_\Box [e] ! m(\overline{e})$ (a) and $e \Box [e'] = e'_\Box [e'] ! m(\overline{e})$. By (3) and Lemma A.6, $\Theta \vdash e' [e] : \tau''$ (b) for some $\tau''$. By applying the induction hypothesis we obtain $\Theta \vdash e'_\Box [e'] : < \tau''$ (c). By (3), (a), (T-ASYNCCALL), and (T-DIRECTCALL), $\Theta \vdash e'_\Box [e] : c$ and $\text{mtype}(c, m) = \overline{\tau} \triangleright \tau'$. With (b), $c = \tau''$. Hence with (c), $\Theta \vdash e'_\Box [e'] : < c$. Thus by (T-SUB), $\Theta \vdash e'_\Box [e'] : c'$, for some $c'$ with $c' <: c$. By the overloading condition (C-OVERRIDEOK), we obtain $\text{mtype}(c', m) = \overline{\tau} \triangleright \tau'$. Thus (T-ASYNCCALL) can be applied, resulting in $\Theta \vdash e'_\Box [e'] ! m(\overline{e}) : \tau'$.

Case $e \Box = v! m(\overline{v}, e'_\Box, \overline{e})$. This case is similar to the case above, but uses the argument that an argument type does not affect the method return type, due to the fact that overloading is forbidden by condition (C-NOOVERLOAD).

Case $e \Box = e'_\Box . m(\overline{e}, e', \overline{e})$. Equivalent to the asynchronous call.

Case $e \Box = v.m(\overline{v}, e'_\Box, \overline{e})$. Equivalent to the asynchronous call.

Case $e \Box = e'_\Box . \text{get}$. Hence $e \Box [e] = e'_\Box [e] . \text{get}$ and $e \Box [e'] = e'_\Box [e'] . \text{get}$. By (3) and (T-FUTGET), $\Theta \vdash e'_\Box [e] : \text{f}(\tau'')$. By the induction hypothesis we obtain $\Theta \vdash e'_\Box [e'] : < \text{f}(\tau'')$. By (T-FUTGET) and (S-FUT) we conclude $\Theta \vdash e'_\Box [e'] . \text{get} : < \tau''$.

Case $e \Box = e'_\Box . \text{await}$. Equivalent to the case above.

Case $e \Box = e'_\Box . \text{resolve } e''$. Hence we obtain $e \Box [e] = e'_\Box [e] . \text{resolve } e''$ and $e \Box [e'] = e'_\Box [e'] . \text{resolve } e''$. By (3) and Lemma A.6, $\Theta \vdash e'_\Box [e'] : \tau''$ for some $\tau''$. By the induction hypothesis we obtain $\Theta \vdash e'_\Box [e'] : \overline{\tau''}$ with $\tau''' <: \tau''$ (a). By (3) and (T-PROMRESOLVE), we conclude $\tau'' = p(\tau')$ and $\Theta \vdash e'' : < \tau'$. As a promise type cannot have any proper subtype, we obtain with (a) that $\tau''' = \tau''$. Thus $\Theta \vdash e'_\Box [e'] : \tau'$.

Case $e \Box = v . \text{resolve } e'$. Immediately by the induction hypothesis and the rule (T-PROMRESOLVE).

Case $e \Box = e'_\Box . \text{fut}$. Analogous to the case $e'_\Box . \text{resolve } e''$. □

Lemma A.8 (Copy Typing) The copying of objects preserves typing. Let $K = K' \cup k \cup k'$. Assume $K \vdash \Sigma$ and $\Sigma \vdash K$. Assume $\emptyset ; \Sigma \vdash v : \tau$. Let $(O'', v') = \text{copy}(id_k, O_k, v, id_{k'}, O_{k'})$. Then there exists a $\Sigma'$ with $\Sigma \subseteq \Sigma'$, such that

a) $\Sigma'; id_{k'} \vdash v''$ and
Appendix A Core Calculus – Details and Proof

\[
b) \quad \emptyset; \Sigma' \vdash e \nu' : \tau
\]

Proof. As the copy function does not introduce new types, a corresponding \(\Sigma'\) can be constructed by simply extending it with the types of the copied identifiers. \(\Box\)

A.2.4 Proof of the Preservation Lemma

This subsection provides a detailed proof of Lemma 3.4.

Proof. The proof is by case analysis on the reduction rules. For the cobox-local rules, we implicitly apply the congruence rule. For easier reference, we repeat the assumptions and goals of the lemma.

Assumptions.

\[
\begin{align*}
dom(\Sigma) &= rdom(K_n) \quad (A1a) \\
\Sigma &\vdash^{\star} K_n \quad (A1b) \\
\Sigma &\vdash K_n \quad (A1) \\
K_n &\rightarrow K_{n+1} \quad (A2)
\end{align*}
\]

Goals.

\[
\begin{align*}
\forall r \in \dom(\Sigma). \Sigma(r) &= \Sigma'(r) \quad (G1) \\
dom(\Sigma') &= rdom(K_{n+1}) \quad (G2a) \\
\Sigma' &\vdash^{\star} K_{n+1} \quad (G2b) \\
\Sigma' &\vdash K_{n+1} \quad (G2)
\end{align*}
\]

Case (R-NewCoBox)

\[
coboxcl(c) \quad \kappa'_b \text{ fresh} \\
b' = b(\kappa'_b, \{o(t, c, init(c))\}, \emptyset, \bullet) \\
t' = \tau(e_{\Box}[\kappa'_b, t]) \\
K \cup b(\kappa_b, O, T, \bar{t} \cdot \tau(e_{\Box}[\text{new } c])) \\
\rightarrow K \cup b(\kappa_b, O, T, \bar{t} \cdot t') \cup b'
\]

By (A1b), \(\Sigma \vdash K_b\), and hence \(\emptyset; \Sigma \vdash e_{\Box}[\text{new } c] : \tau\) (1), for some \(\tau\). By Lemma A.6 and (T-New), \(\emptyset; \Sigma \vdash e_{\Box}[\text{new } c] : c\). Let \(\Sigma' = \Sigma, \kappa'_b, t : c\). Hence we get (G1) because \(\kappa'_b\) is fresh and thus \(\kappa'_b, t \notin \dom(\Sigma)\). We obtain (G2a) because of (A1a) and the fact that \(b'\) has an object which is referenced by \(\kappa'_b, t\). By (T-Ref), \(\emptyset; \Sigma' \vdash e_{\Box}[\kappa'_b, t] : \tau\). With (1), Lemma A.7, and Lemma 1.5a, we can conclude \(\emptyset; \Sigma' \vdash e_{\Box}[\kappa'_b, t] : \preceq \tau\). Thus \(\Sigma' \vdash b_{n+1}\). Finally, we have to show \(\Sigma' \vdash b'\). This is by \(\Sigma'; \kappa'_b \vdash o(t, c, init(c))\), which is satisfied. Thus we obtain (G2b) and finally (G2).

Case (R-NewObjFar) Similar to the case above.
A.2 Type Soundness

Case (R-PromNew)

\[ K' = K \cup \rho(\kappa_p, \emptyset, \epsilon) \]

\[ K \cup B(\kappa_b, O, T, e) \xrightarrow{b_n} K' \cup B(\kappa_b, O, T, e) \xrightarrow{b_{n+1}} \]

By (A1b), \( \Sigma \vdash_k b_n \), and hence \( \emptyset; \Sigma \vdash_{e} \tau \) by (T-CoBox), Lemma A.6, and (T-PromNew). Let \( \Sigma' = \Sigma, \kappa_p : \rho(\tau) \). Hence (G1) and (G2a), because \( \kappa_p \) is fresh and (A1a). In addition, we obtain \( \emptyset; \Sigma \vdash_{e} \kappa_p : \rho(\tau) \). With Lemma A.7, Lemma 1.5a, and (T-CoBox), we get \( \Sigma' \vdash_k b_{n+1} \). In addition, \( \Sigma' \vdash_k \rho(\kappa_p, \emptyset, \epsilon) \) holds. Hence we obtain (G2b) and thus (G2).

Case (R-PromResolve)

\[ (O', v') = copy(\kappa_b, O, v, \kappa_p, \emptyset) \]

\[ K = K' \cup \rho(\kappa_p, \emptyset, \epsilon) \]

\[ K'' = K' \cup \rho(\kappa_p, O', v') \]

\[ t = \tau(e_\Box[null]) \]

\[ K \cup B(\kappa_b, O, T, e) \xrightarrow{b_n} K' \cup B(\kappa_b, O, T, e) \xrightarrow{b_{n+1}} \]

By (A1b), \( \Sigma \vdash_k b_n \), and hence \( \Sigma; \kappa_p \vdash_o O \) (1) and \( \emptyset; \Sigma \vdash_{e} \tau \) by (A1a), (3), and Lemma A.8, there exists a \( \Sigma' \) with \( \Sigma \subseteq \Sigma' \) such that \( \emptyset; \Sigma \vdash_{e} \tau \) and \( \Sigma'; \kappa_p \vdash_o O' \). Hence, \( \Sigma' \vdash_k \rho(\kappa_p, O', v') \). By (4) and Lemma A.5, \( \Sigma' \vdash_k b_{n+1} \). Hence we obtain (G2a) and (G2b) and thus (G2).

Case (R-PromFut)

\[ \rho(\kappa_p, \cdot, \cdot) \in K \quad t \notin oids(O) \quad O = \tau(\kappa_p, \epsilon) \quad t = \tau(e_\Box[\kappa_b,t]) \]

\[ K \cup B(\kappa_b, O, T, e) \xrightarrow{b_n} K' \cup B(\kappa_b, O \cup \rho(\kappa_p, \kappa_p, \kappa_p) \cup \rho(\kappa_p, \kappa_p, \kappa_p)). \]

By (A1b), \( \Sigma \vdash_k b_n \), and hence \( \Sigma; \kappa_p \vdash_o O \) (1). By (T-PromFut), \( \emptyset; \Sigma \vdash_{e} \tau \) and \( \emptyset; \Sigma \vdash_{e} \kappa_p : \rho(\tau) \) for some \( \tau \). Let \( \Sigma' = \Sigma, \kappa_b, \tau : \tau \). Then (G1), because \( t \notin oids(O) \). We can then conclude that \( \Sigma' ; \kappa_b \vdash_o \tau(\kappa_p, \epsilon) \) (2), and also that \( \emptyset; \Sigma' \vdash_{e} \kappa_b, \tau : \tau \). Hence it can be derived that \( \Sigma' \vdash_k t \). With (1) and (2) we get \( \Sigma' \vdash_k b_{n+1} \) and thus we obtain (G2b). By (A1a), (G2a). And thus we get (G2).

Case (R-FutResolve) The proof is similar to the case of rule (R-PromResolve) and is mainly based on Lemma A.8.
Case \((R\text{-asyncCallLocal})\)

\[
\begin{align*}
\rho &= \rho(\kappa_p, \varnothing, e) \\
e &= \text{mexpr}(c, \kappa_b, \tau, m, \overline{v}) \\
t' &= \tau(\kappa_p, \text{fresh} e) \\
K \cup b(\kappa_b, O, T, \bar{\tau} \cdot \tau(e_{\square}[\kappa_b, \text{fut}][\bar{v}])) \rightarrow K \cup b(\kappa_b, O, T, \bar{\tau} \cdot \tau(e_{\square}[\kappa_b, \text{fut}][\bar{v}]) \cup \rho)
\end{align*}
\]

By (A1b), \(\Sigma \vdash_k b_n\), and hence \(\Sigma \vdash_0 O(1)\) and \(\varnothing; \Sigma \vdash_k \kappa_b, \tau, m, \overline{v} : \tau(2)\), for some \(\tau\). By (T-asyncCall): \(\varnothing; \Sigma \vdash_k \kappa_b, \tau, m, \overline{v} : \tau(3)\), \(\varnothing; \Sigma \vdash_0 \overline{v} : \tau'(4)\), \(\text{mtype}(c, m) = \overline{v} \Rightarrow \tau'\) (5), and \(\tau = \tau'(6)\). Let \(\tau' = \tau, \kappa_p : \tau(\tau')\). Hence (G1), because of (A1a) and the fact that \(\kappa_p\) is fresh. As no other new reference is introduced we conclude (G2a). Furthermore, we can obtain \(\varnothing; \Sigma' \vdash_k \kappa_p, \text{fut} : \rho(\tau')\) and hence \(\Sigma' \vdash_1 t'\) (6), by Lemma A.7 and (T-Tsk). By (T-Promise), \(\Sigma' \vdash_k p(7)\). We finally have to show \(\Sigma' \vdash_k \tau''\) in order to obtain \(\Sigma' \vdash_k b_{n+1}\). As \(\tau'' = \tau(\kappa_p, \text{resolve} e)\) and \(\varnothing; \Sigma' \vdash_k \kappa_p : \rho(\tau')\), we have to show that \(\varnothing; \Sigma' \vdash_k e : \tau'\). By the definition of \(\text{mexpr}\) (cf. Table 3.2), we get \(e = [r / \text{this}, \overline{v}/\overline{x}] e', \) where \(\overline{x} e' = mbody(c, m)\). By (1), \(\varnothing; \Sigma \vdash_k \kappa_b, \tau, c\). By (5) and (T-Method), this: \(c, \overline{x} : \tau'\); \(\varnothing; \Sigma \vdash_0 \overline{v} : \tau'\). With (4) and Lemma A.3 we obtain \(\varnothing; \Sigma' \vdash_1 e : \tau'\). By (T-PromiseResolve) and (T-Tsk), \(\Sigma' \vdash_1 t''\). With (6) we get \(\Sigma' \vdash_k b_{n+1}\). Thus (G2b), and hence (G2).

Case \((R\text{-asyncCallFar})\) The proof is similar to the case above, but in addition involves the treatment of the copying of parameters, which is similar to case \((R\text{-PromiseResolve})\).

Case \((R\text{-let})\)

\[
\begin{align*}
K \cup b(\kappa_b, O, T, \bar{\tau} \cdot \tau(\kappa_p, e_{\square}([x = \nu \in e]))) \rightarrow K \cup b(\kappa_b, O, T, \bar{\tau} \cdot \tau(\kappa_p, e_{\square}([\nu/x]e)))
\end{align*}
\]

By (A1b), \(\Sigma \vdash_k b_n\) and hence \(\varnothing; \Sigma \vdash_e e : \tau\), for some \(\tau\). By (T-Let), \(\varnothing; \Sigma \vdash_0 \nu : \tau'\) and \(\varnothing, x : \tau'; \Sigma \vdash_e e : \tau\). By Lemma A.3, \(\varnothing; \Sigma \vdash_0 [\nu/x]e : \tau\). Hence \(\varnothing; \Sigma \vdash_e e_{n+1} : \tau\). Hence \(\Sigma \vdash_k b_{n+1}\) and thus (G2b). (G1) and (G2a), can then be obtained immediately as \(\Sigma' = \Sigma\) and \(\text{rdom}(K_n) = \text{rdom}(K_{n+1})\).

Case \((R\text{-newObjLocal})\)

\[
\begin{align*}
e &= \text{new} c \lor e = \text{new} c \text{ in } \kappa_b, \tau' \\
\neg \text{coboxcl}(c) &\lor e \notin \text{oids}(O) \\
o = o(\tau, c, \text{init}(c)) \\
K \cup b(\kappa_b, O, T, \bar{\tau} \cdot \tau(e_{\square}[e])) \rightarrow K \cup b(\kappa_b, O \cup o, T, \bar{\tau} \cdot \tau(e_{\square}[\kappa_b, \tau]))
\end{align*}
\]

By (A1b), \(\Sigma \vdash_k b_n\) and hence \(\varnothing; \Sigma \vdash_e e : \tau\), for some \(\tau\). By (T-New), or (T-NewIn), depending on the form of \(e\), we obtain in both cases that \(\tau = c\). Let \(\Sigma' = \Sigma, \kappa_b, \tau : c\). Hence (G1), as \(\kappa_b, \tau \notin \text{dom}(\Sigma)\), and (G2a) by (A1a) as
\[ rdom(K_{n+1}) = rdom(K_n) \cup \{ \kappa_b.t \}. \] By (T-Obj), \( \Sigma' \vdash_0 \sigma \). By (T-Ref), \( \emptyset; \Sigma' \vdash e_b \kappa_b.t : \tau \). Hence by Lemma A.7 and (T-CoBox), we get (G2b). Thus we obtain (G2).

**Case** (R-DirectCall) Similar to case (R-AsyncCallLocal).

**Case** (R-FieldSelect)

\[
\begin{align*}
o(t, c, \bar{v}) \in O & \quad \text{fields}(c) = \tau f \\
K \cup B(\kappa_b, O, T, \bar{e} \cdot \tau(e_{\Box}[\kappa_b.t.f_i])) & \rightarrow K \cup B(\kappa_b, O, T, \bar{e} \cdot \tau(e_{\Box}[v_i])) \\
\end{align*}
\]

By (A1b), \( \Sigma \vdash_k b_n \). Hence \( \Sigma \vdash_0 O \) (1) and \( \emptyset; \Sigma \vdash e_b \kappa_b.t.f_i : \tau_i \). By (T-FieldSelect), \( \emptyset; \Sigma \vdash e_b \kappa_b.t : c \) and \( \text{fields}(c) = \tau f \). By (1) and (T-Obj), \( \emptyset; \Sigma \vdash e_b \bar{v} :< \bar{\tau} \). Thus \( \emptyset; \Sigma \vdash e_b v_i :< \tau_i \). Hence \( \Sigma' \vdash_k b_{n+1} \) and thus (G2b). (G1) and (G2a) are immediate as \( \Sigma' = \Sigma \) and \( rdom(K_n) = rdom(K_{n+1}) \).

**Case** (R-FieldUpdate)

\[
\begin{align*}
o = O' \cup o(t, c, \bar{v}) & \quad O'' = O' \cup o(t, c_i[v/v_i] \bar{v}) \\
K \cup B(\kappa_b, O, T, \bar{e} \cdot \tau(e_{\Box}[\kappa_b.t.f_i = v])) & \rightarrow K \cup B(\kappa_b, O'', T, \bar{e} \cdot \tau(e_{\Box}[v])) \\
\end{align*}
\]

By (A1b), \( \Sigma \vdash_k b_n \). Hence \( \Sigma \vdash_0 O \) (1) and \( \emptyset; \Sigma \vdash e_b \kappa_b.t.f = v : \tau \) (2). By (T-Obj) and (1), \( \emptyset; \Sigma \vdash e_b \kappa_b.t : c \) and \( \emptyset; \Sigma \vdash e_b \bar{v} :< \bar{\tau} \), where \( \text{fields}(c) = \tau f \). By (T-FieldUpdate), \( \emptyset; \Sigma \vdash e_b [v/v_i] \bar{v} :< \bar{\tau} \). Hence \( \emptyset; \Sigma \vdash e_b O'' \). Hence we obtain \( \Sigma' \vdash_k b_{n+1} \) and thus (G2b). (G1) and (G2a) are immediate as \( \Sigma' = \Sigma \) and \( rdom(K_n) = rdom(K_{n+1}) \).

**Case** (R-Yield) Immediate as yield and null can be typed to any type.

**Case** (R-FutAwait)

\[
K \cup B(\kappa_b, O, T, \bar{e} \cdot \tau(e_{\Box}[r.await])) \rightarrow K \cup B(\kappa_b, O, T \cup \tau(e_{\Box}[r.get]), \bar{T})
\]

Immediate, as the operators await and get are typed identically and the overall typing of the coobox is not affected by moving the active task to the suspend set.

**Case** (R-TaskResume)

\[
\begin{align*}
n(t, \_ , \bar{v}) & \in O \\
K \cup B(\kappa_b, O, T \cup \tau(e_{\Box}[\kappa_b.t.get]), \bar{\tau}) & \rightarrow K \cup B(\kappa_b, O, T, \tau(e_{\Box}[\bar{v}]) \cdot \bar{\tau}) \\
\end{align*}
\]

By (A1b), \( \Sigma \vdash_k b_n \). Hence \( \Sigma \vdash_0 O \) and \( \emptyset; \Sigma \vdash e_b \kappa_b.t.get : \tau \). By (T-Fut) and (T-FutGet), \( \emptyset; \Sigma \vdash_0 \kappa_b.t : \tau \) and \( \emptyset; \Sigma \vdash_0 v :< \tau \). Hence \( \Sigma' \vdash_k b_{n+1} \) and thus (G2).
Case (R-TASK-TERMINATE) Immediate.

Case (R-FUTGET) Like case (R-TASKRESUME).

A.2.5 Proof of the Progress Lemma

This subsection provides the proof of the Progress Lemma (Lemma 3.6).

Proof. By the fact that $K_n$ is not terminal, i.e., $\neg \text{terminal}(K_n)$, there either exists a $b \in K_n$ with $\neg \text{inactive}(K_n, b)$ (A1a), or there exists a resolvable future in $K_n$, i.e., $b, \rho \in K_n$ with $\rho = \rho(\kappa_p, O, v) \land \rho(t, \kappa_p, e) \in O_b$ (A1b). If (A1b) holds, then rule (R-FUTRESOLVE) can always be applied and hence there is a $K_{n+1}$ with $K_n \rightarrow K_{n+1}$. We thus assume that only (A1a) holds. Hence, by the definition of an inactive cobox, there either exists a task $t \in T_b$ with $\neg \text{blockedtsk}(b, t)$ (A1a1), or there exists an active task $t$ and $t$ is neither broken nor blocked (A1a2). We first assume (A1a1). Hence, by the definition of a blocked task, either $t \neq \tau(e[\kappa_b, t, \text{get}])$ or $\tau(t, _, v) \notin O_b$. By Lemma 3.3, all suspended tasks have the form $t = \tau(e[\kappa_b, t, \text{get}])$, thus only the latter condition may be true. By $\Sigma \models K_n$ there must be a future $o \in O_b$ with $o = \tau(t, _, v)$ for some $v$. But then rule (R-TASKRESUME) can be applied and thus there is a $K_{n+1}$ with $K_n \rightarrow K_{n+1}$.

We now only assume (A1a2), i.e., there exists an active task, which is neither broken nor blocked. We thus assume that $K_n = K' \cup b$, where $b = b(\kappa_b, O, T, t \cdot t)$. To show that there is always a rule that can be applied, we do a structural induction on the form of expression $e$ of the active task $t$.

**Assumptions.** The following assumptions are given.

$$K_n = K' \cup b(\kappa_b, O, T, t \cdot t)$$

$$t = \tau(e) \quad \neg \text{brokenresolve}(K_n, e)$$

$$\neg \text{nullacc}(e) \quad \neg \text{fdaccess}(\kappa_b, e)$$

$$\neg \text{brokenexpr}(K_n, \kappa_b, t) \quad \neg \text{blockedsk}(b, t)$$

$$\neg \text{inactive}(K_n, b)$$

$$\emptyset; \Sigma \vdash e$$

$$\Sigma; \kappa_b \vdash^* O \quad \Sigma \vdash^* t \quad \Sigma \vdash^* \tau(e)$$

$$\Sigma \vdash^* K'$$

$$\Sigma \vdash^* b(\kappa_b, O, T, t \cdot \tau(e))$$

$$\Sigma \vdash^* K_n$$

$$\Sigma \models K_n$$

174
Goals.

\[ K_n \rightarrow K_{n+1} (G) \]

**Case** \( e = e[\Box[v]] \). If \( e[\Box] = \Box \), then \( e = v \), and rule (R-TASK-TERMINATE) can be applied. If \( e[\Box] \neq \Box \) then there is some \( e' \neq e[\Box] \) and \( e' \) with \( e = e'[\Box[e']] \) and we can apply the induction hypothesis.

**Case** \( e = e[\Box[x]] \). This case cannot happen, because \( e \) is typed under an empty variable typing. And thus there cannot be free variables in \( e \).

**Case** \( e = e[\Box[let \ x = v \ in \ e]] \). In this case rule (R-LET) can be applied, which has no preconditions.

**Case** \( e = e[\Box[new \ c]] \). Depending on the kind of \( c \), two rules can be applied. If \( c \) is a cobox class, then (R-NEWCOBOX) can be applied. It has no further preconditions. If \( c \) is not a cobox class, then rule (R-NEWOBJLOCAL) can be applied.

**Case** \( e = e[\Box[new \ c \ in \ v]] \). If \( v = \text{null} \), then \( e \) is broken, which is not allowed by the assumptions. Hence \( v \neq \text{null} \), i.e., \( v = r \), for some \( r \). By \( \emptyset; \Sigma \vdash e \) and (T-NEWIN), \( \emptyset; \Sigma \vdash (r : c \ (1)) \) and \( \text{plaincl}(c) \). Hence \( r \neq \kappa_p \) and thus \( r = \kappa'_b \cdot t \), for some \( \kappa'_b \) and \( t \). There are now two cases. If \( \kappa'_b = \kappa_b \), \( r \) is a local reference, then rule (R-NEWOBJLOCAL) can be applied. If \( \kappa'_b \neq \kappa_b \), \( r \) is a far reference, then only rule (R-NEWOBJFAR) can be applied. That rule, however, requires that there exists a \( b' \in K' \), with \( \text{id}_{b'} = \kappa_b \). By (1), \( \Sigma(\kappa'_b \cdot t) = c \). With (A2a) we can conclude that there must be a cobox with identifier \( \kappa'_b \) that has an object with identifier \( t \). Thus rule (R-NEWOBJFAR) can be applied.

**Case** \( e = e[\Box[v.f]] \). As \( e \) may not be broken, \( v = r \), for some \( r \). In addition, \( r.f \) cannot be a far direct access and thus \( r = \kappa_b \cdot t \), for some \( t \). By (T-FIELDSELECT), \( \emptyset; \Sigma \vdash r : c \) and \( \text{fields}(c) = \tau \cdot \bar{f} \) and there exists a \( f_i \in \bar{f} \) with \( f_i = f \). Hence rule (R-FIELDSELECT) can be applied.

**Case** \( e = e[\Box[v.f = v']] \). Like the case above.

**Case** \( e = e[\Box[yield]] \). In this case, rule (R-YIELD) can be applied, as it has no other preconditions.

**Case** \( e = e[\Box[v.await]] \). As \( e \) cannot be broken, \( v = r \), for some \( r = \kappa'_b \cdot t \). By (T-FUTAWAIT), \( \emptyset; \Sigma \vdash r : f(\tau) \). By Lemma 3.3, \( r \) is a local reference, i.e., \( \kappa'_b = \kappa_b \), because there cannot exist any far references to futures (WF-TRANSFER). Hence rule (R-FUTAWAIT) can be applied.

**Case** \( e = e[\Box[v.get]] \). As \( e \) is not broken, \( v = r \), for some \( r = \kappa'_b \cdot t \). By (T-FUTGET), \( \emptyset; \Sigma \vdash r : f(\tau) \). By Lemma 3.3, \( r \) is a local reference, i.e., \( \kappa'_b = \kappa_b \), because there cannot exist any far references to futures (WF-TRANSFER). By (A2a) we
conclude that there has to exist a future $o \in O$, with $o = \rho(t, \kappa_p, \_$. By the assumption that task $t$ is not blocked, we obtain $o \neq \rho(t, \kappa_p, \epsilon)$. Hence there must be some $v$ with $o = \rho(t, \kappa_p, v)$. Thus rule (R-TASKRESUME) can be applied.

**Case** $e = e_{\square}[promise \tau]$. In this case rule (R-PROMNEW) can be applied, which has no other preconditions.

**Case** $e = e_{\square}[v.resolve v']$ By the assumption that $e$ is not broken, $v \neq \text{null}$. Hence, with (T-PROMRESOLVE), $v = \kappa_p$, for some $\kappa_p$. By $\neg\text{brokenresolve}(K_n, e)$, there must exist a $\rho \in K_n$ with $\rho = \rho(\kappa_p, \emptyset, \epsilon)$. Thus rule (R-PROMRESOLVE) can be applied.

**Case** $e = e_{\square}[v.fut]$ By the assumption that $e$ is not broken, $v = r$, for some $r$. By rule (T-PROMFUT), $\emptyset; \Sigma \vdash e : \rho(\tau)$, for some $\tau$. Hence, $r = \kappa_p$, for some $\kappa_p$. By (A2a), there has to be some $\rho(\kappa_p, \_, \_)$ $\in K_n$. Thus rule (R-PROMFUT) can be applied.

**Case** $e = e_{\square}[v.m(\nu)]$. As $e$ may not be broken, $v = r$, for some $r$. In addition, $r.m(\nu)$ cannot be a far direct access and $r = \kappa_b.t$, for some $t$. By (T-DIRECTCALL), $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$. Hence there exists a method $m$ in class $c$ and function $mexpr$ is defined. Thus rule (R-DIRECTCALL) can be applied.

**Case** $e = e_{\square}[v!m(\nu)]$. This case is similar to the case above, with the difference that an asynchronous call does not require a local references as target. Instead, the form of the target reference $r$ decides whether a local or a far call is executed. If $r$ is local, the proof is analogous to the proof of the direct call. So we assume that $r$ is a far reference, i.e., $r = \kappa'_b.t$, where $\kappa'_b \neq \kappa_b$. By (T-ASYNCALL), $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$ and $\emptyset; \Sigma \vdash e : \rho(\tau)$. Finally, rule (R-ASYNCALLFAR) can be applied with similar arguments as in the case above. Note that by Property 3.1, copy can always be applied.
This chapter presents the implementation of the dynamic semantics of JCoBoxC in the rewriting logic framework Maude [CDE+07]. Maude is in particular suited for defining the semantics of concurrent programming languages, as rewriting theories are evaluated concurrently.

We refer to the Maude implementation of JCoBoxC by JCoBoxM in the following. JCoBoxM is not designed to be an efficient implementation of JCoBoxC, instead, it is designed to resemble the JCoBoxC formalization as close as possible. Having an implementation in Maude allows programs written in JCoBoxM to be executed by Maude. This makes it possible to easily test and experiment with the JCoBoxM semantics and programs written in JCoBoxM. JCoBoxM can thus be seen as a way to evaluate the JCoBoxC semantics. Maude can also be used as an LTL model checker to analyze programs written in JCoBoxM, to find deadlocks, for example.

JCoBoxM consists of a syntax definition, several auxiliary functions, the definition of substitution and evaluation contexts, and a dynamic semantics. A type system is not implemented. In this chapter, we present the semantic entities, evaluations contexts, and the dynamic semantics of JCoBoxM.

### B.1 Semantic Entities

The dynamic entities are defined by a set of sorts and corresponding constructors. We left out the definition of several auxiliary functions.

#### B.1.1 Sorts

Listing B.1 presents the SEMANTIC-SORTS module, that defines the sorts of the semantic entities. With some slight differences, the sorts are equal to the syntactic categories of the semantic entities of JCoBoxC (cf. Table 3.9, page 57). Component identifiers of JCoBoxM are not further distinguished into promise identifiers and cobox identifiers,
fmod SEMANTICS-SORTS is
  protecting NAT .
  protecting SUBSTITUTION .
  extending SYNTAX-EXPR-SIG .

  sorts ObjRef .
  subsort ObjRef < Value .

  sorts CompId ObjId .
  subsorts Nat < ObjId CompId .
  subsort CompId < Value .

  sorts OptValue .
  subsort Value < OptValue .

  sorts Task Object CoBox Prom Config Comp .
  subsort CoBox Prom < Comp .

  sorts ObjSet TaskSet TaskList CompSet .
  subsort Object < ObjSet .
  subsort Task < TaskSet .
  subsort Task < TaskList .
  subsort Comp < CompSet .
endfm

Listing B.1: The sorts of the semantic entities of JCoBox£.

which is not necessarily needed by the semantics, but technically complicates the Maude implementation. Object and component identifiers are super sorts of the Maude sort Nat, which represents natural numbers. This essentially means that every natural number can be used as an object identifier and a component identifier.

B.1.2 Notations

JCoBox£ uses different notations in some cases compared to JCoBox£. The main differences are explained in the following.

Sets

Sets are defined in Maude by defining a constructor for the empty set, and a union operator that is commutative and associative. In addition, an equation has to be given to realize idempotency. The union operator for all sets of JCoBox£ is simply a space. For example, the JCoBox£ term $K \ K'$ corresponds to $K \cup K'$ in JCoBox£. Note that if a term is a left hand side of Maude’s matching equation :=, e.g., $K \ K' := K''$, the union operator actually corresponds to the operator $\cup$. 

178
B.1 Semantic Entities

Sequences

Sequences are defined like sets by defining an empty list constructor and a concatenation operator, which, however, is only associative. In JCoBox\textsuperscript{M} the concatenation operator is a colon :\textsuperscript{C}, and corresponds to the operators • and · used in JCoBox\textsuperscript{C}.

Maps

JCoBox\textsuperscript{C} uses the built-in MAP module of Maude to realize maps from variables to values. Given a map vm, the value for key x is obtained by vm[x]. To update vm with a map from x to v, one writes vm,x |-> v.

B.1.3 Constructors

Listing B.2 shows the SEMANTICS-CONSTR module of JCoBox\textsuperscript{M} that defines the constructors for the semantic entities. They are nearly identical to the constructors used in JCoBox\textsuperscript{C}, with some exceptions. The state of objects is expressed by a map from variable names to values, instead of a sequence of values. Using an explicit map makes it easier in Maude. Also note that field names are unified with variable names in JCoBox\textsuperscript{M}. The second difference is that in Maude the underlying program must always be explicitly represented. Thus, configurations in JCoBox\textsuperscript{M} are pairs, consisting of a program and a set of components.

B.1.4 Substitution

The substitution of expressions is defined in module SUBSTITUTION. It is a rather straightforward definition and corresponds to Definition A.1, so we do not show it here. We only show the corresponding operators:

\begin{verbatim}
op subst : Expr Expr Expr -> Expr .
op [/_/]_ : Expr Expr Expr -> Expr .
eq [ es / ep ] e = subst(es,ep,e) .
\end{verbatim}

The actual definition is done by operator subst. The operator [/_/]\_ is just syntactic sugar.

B.1.5 Evaluation Contexts

Evaluation contexts are realized by module CONTEXT, which is shown in Listing B.3. It defines two new constructors for expressions, namely hole and \_\_[\textvisiblespace}. The operator toCxt takes an expression and turns it into the corresponding evaluation context. This essentially means that it recursively applies toCxt to subexpressions until not further possible. Also important is the equation e [ v ] = .., which creates the next evaluation context when the current expression of the evaluation context is a value.
fmod SEMANTICS-CONSTR is
  including SEMANTICS-SORTS .
  protecting CONTEXT .
  protecting SYNTAX-PROGRAM-HELPER .

  −−− map from variable names to values
  protecting MAP(VarName,Value) * (sort Map(VarName,Value) to ValueMap) .

  op oref : CompId ObjId -> ObjRef [ctor] .          −−− object references
  op noVal : -> OptValue [ctor] .                     −−− corresponds to ε
  op obj : ObjId ClassName ValueMap -> Object [ctor] . −−− objects
  op fut : ObjId CompId OptValue -> Object [ctor] .   −−− futures
  op tsk : Expr -> Task [ctor] .                      −−− tasks
  op cb : CompId ObjSet TaskSet TaskList -> CoBox    −−− coboxes
         [ctor format (n d)] .
  op prom : CompId ObjSet OptValue -> Prom [ctor] .  −−− promises
  op conf : Program CompSet -> Config                −−− configurations
         [ctor frozen (1)] .

  op noObj : -> ObjSet [ctor] .                       −−− empty object set
  op __ : ObjSet ObjSet -> ObjSet                   −−− union operator
         [comm assoc id: noObj] .
  var o : Object .
  eq o o = o .                                        −−− idempotency

  op noComp : -> CompSet [ctor] .                     −−− empty component set
  op __ : CompSet CompSet -> CompSet                 [comm assoc id: noComp] .
  var k : Comp .
  eq k k = k .

  op noTsks : -> TaskSet [ctor] .                     −−− empty task set
  op __ : TaskSet TaskSet -> TaskSet                 [comm assoc id: noTsks] .
  var t : Task .
  eq t t = t .

  op emptyTskList : -> TaskList [ctor] .             −−− empty task list
  op __ : TaskList TaskList -> TaskList             −−− list concatenation
         [assoc id: emptyTskList] .
endfm

Listing B.2: The SEMANTICS-CONSTR module of JCoBoxm.
B.1 Semantic Entities

--- turn expressions into a contexts

<table>
<thead>
<tr>
<th>op</th>
<th>toCxt</th>
<th>Expr</th>
<th>-&gt;</th>
<th>Expr</th>
</tr>
</thead>
</table>
| eq  | toCxt(e1 [ e2 ] ) = e1 [ e2 ] .
| ceq | toCxt(let x = nve in e ) = (let x = e1 in e ) [ e2 ]
|     | if e1 [ e2 ] := toCxt(nve) .
| ceq | toCxt(new c in nve) = (new c in e1) [ e2 ]
|     | if e1 [ e2 ] := toCxt(nve) .
| ceq | toCxt(nve . x) = (e1 . x) [ e2 ]
|     | if e1 [ e2 ] := toCxt(nve) .
| ceq | toCxt(nve . x = e) = (e1 . x = e) [ e2 ]
|     | if e1 [ e2 ] := toCxt(nve) .
| ceq | toCxt(v . x = nve) = (v . x = e1) [ e2 ]
|     | if e1 [ e2 ] := toCxt(nve) .
| ceq | toCxt(nve . m ( E ) ) = (e1 . m ( E )) [ e2 ]
|     | if e1 [ e2 ] := toCxt(nve) .
| ceq | toCxt(v . m ( (V, nve, E) ) ) = (v . m ( (V, e1, E) )) [ e2 ]
|     | if e1 [ e2 ] := toCxt(nve) .
| ceq | toCxt(nve . get) = (e1 . get) [ e2 ]
|     | if e1 [ e2 ] := toCxt(nve) .
| ceq | toCxt(nve . await) = (e1 . await) [ e2 ]
|     | if e1 [ e2 ] := toCxt(nve) .
| ceq | toCxt(nve . resolve e) = (e1 . resolve e) [ e2 ]
|     | if e1 [ e2 ] := toCxt(nve) .
| ceq | toCxt(v . resolve nve) = (v . resolve e1) [ e2 ]
|     | if e1 [ e2 ] := toCxt(nve) .
| ceq | toCxt(nve . await) = (e1 . await) [ e2 ]
|     | if e1 [ e2 ] := toCxt(nve) .
| ceq | toCxt(nve ! m ( E ) ) = (e1 ! m ( E )) [ e2 ]
|     | if e1 [ e2 ] := toCxt(nve) .
| ceq | toCxt(v ! m ( (V, nve, E) ) ) = (v ! m ( (V, e1, E) )) [ e2 ]
|     | if e1 [ e2 ] := toCxt(nve) .
| eq  | toCxt(nve) = hole [ nve ] [ owise ] .
| eq  | toCxt(v) = v .

--- remove empty contexts

eq hole [ v ] = v .

--- push values into holes

| ceq | e [ v ] = toCxt(subst(v,hole,e))
|     | if e /= hole .

--- normalize contexts

| eq  | e [ e‘ [ e” ] ] = subst(e’,hole,e) [ e” ] .
| eq  | e [ e‘ . resolve (e” [ e”’ ] ) ] = subst(e‘ . resolve e” , hole,e) [ e”’ ] .

Listing B.3: The evaluation contexts of JCoBoxm.
B.2 Transition Rules

The transition rules of the operational semantics are split into cobox-local rules and global rules.

B.2.1 Meta-Variables

Maude supports the concept of meta-variables, which allows for a concise formalization of the rules. The meta-variables, which are used by the rules are shown in Listing B.4.

```
vars p : Program.
vars c : ClassName.
vars m : MethName.
vars k k' kp kp' : CompId.
vars i i' : ObjId.
vars O O' O'' O''' : ObjSet.
vars K K' K'' : CompSet.
vars T T' : TaskSet.
vars TL TL' : TaskList.
vars t t' : Task.
vars vm : ValueMap.
vars x : VarName.
vars v v' v'' : Value.
vars v_opt : OptValue.
vars V V' : ValueList.
vars e1 e2 e e' e'' ep ep' : Expr.
vars nve : NExpr.
vars E E' : ExprList.
vars r : ObjRef.
```

Listing B.4: The meta-variables used by the rules of JCoBox©.

B.2.2 CoBox-Local Rules

The cobox-local rules of JCoBox© are shown in Listing B.5. Note that some rules are actually defined on configurations instead of coboxes. The reason is that these rules require the underlying program, which is not present in coboxes. Besides some technicalities, the cobox-local rules of JCoBox© correspond to the rules of JCoBox©.

182
B.2 Transition Rules

**[r] [R-Let]**:
\[
\text{let } x = v \text{ in } e \Rightarrow \text{toCxt}([v / x] e) .
\]

**[c] [R-NewObjLocal]**:
\[
\text{conf}(p, K \text{cb}(k,0,T, TL : tsk( ep [ new c ] )))
\Rightarrow \text{conf}(p, K \text{cb}(k,0 \text{obj}(i,c,\text{init}(p,c)),T,TL : tsk( ep [ oref(k,i) ] )))
\text{if } i := \text{freshObjId}(0) \land \\
\not \text{isCoBoxClass}(p,c) .
\]

**[c] [R-NewObjLocalIn]**:
\[
\text{conf}(p, K \text{cb}(k,0,T, TL : tsk( ep [ new c in oref(k,i')] )))
\Rightarrow \text{conf}(p, K \text{cb}(k,0 \text{obj}(i,c,\text{init}(p,c)),T,TL : tsk( ep [ oref(k,i) ] )))
\text{if } i := \text{freshObjId}(0) \land \\
\not \text{isCoBoxClass}(p,c) .
\]

**[c] [R-DirectCall]**:
\[
\text{conf}(p, K \text{cb}(k,0,T,TL : tsk( ep [ oref(k,i).m(V) ] )))
\Rightarrow \text{conf}(p, K \text{cb}(k,0,T,TL : tsk( ep [ mexpr(p,c,oref(k,i),m,V) ] )))
\text{if } O' \text{obj}(i,c,vm) := O .
\]

**[r] [R-FieldSelect]**:
\[
\text{cb}(k,0 \text{obj}(i,c,vm),T, TL : tsk( ep [ oref(k,i).x ] ))
\Rightarrow \text{cb}(k,0 \text{obj}(i,c,vm),T, TL : tsk( ep [ vm[x] ] )) .
\]

**[r] [R-FieldUpdate]**:
\[
\text{cb}(k,0 \text{obj}(i,c,(vm,x \rightarrow v')),T, TL : tsk( ep [ oref(k,i).x = v''] ))
\Rightarrow \text{cb}(k,0 \text{obj}(i,c,(vm,x \rightarrow v'')),T,TL : tsk( ep [ v''] )) .
\]

**[c] [R-FutGet]**:
\[
\text{cb}(k,0,T, TL : tsk( ep [ oref(k,i).get ] ))
\Rightarrow \text{cb}(k,0,T, TL : tsk( ep [ v ] ))
\text{if } 0' \text{fut}(i,kp,v) := O .
\]

**[r] [R-FutAwait]**:
\[
\text{cb}(k,0,T, TL : tsk( ep [ r.await] ))
\Rightarrow \text{cb}(k,0,T \text{tsk( ep [ r.get ] )}, TL) .
\]

**[c] [R-TaskResume]**:
\[
\text{cb}(k,0,T \text{tsk( ep [ oref(k,i).get ] )}, TL)
\Rightarrow \text{cb}(k,0,T \text{tsk( ep [ v ] )}, TL)
\text{if } 0' \text{fut}(i,kp,v) := 0 .
\]

**[r] [Yield]**:
\[
\text{cb}(k,0,T, TL : tsk( ep [ yield ] ))
\Rightarrow \text{cb}(k,0,T, tsk( ep [ null ] ) : TL) .
\]

**[r] [R-TaskTerminate]**:
\[
\text{cb}(k,0,T, TL : tsk( null )) \Rightarrow \text{cb}(k,0,T, TL) .
\]

---

Listing B.5: The coobox-local rules of JCoBox™.
B.2.3 Global Rules

The global rules of JCoBox$^M$ are given in Listing B.6. The promise expression does not take a type argument in JCoBox$^M$ as this is not needed by the dynamic semantics. A congruence rule is not needed as Maude automatically applies rules to subterms where possible and not explicitly prevented. Otherwise the global rules of JCoBox$^M$ are semantically equal to the rules of JCoBox$^C$. 
B.2 Transition Rules

Listing B.6: The global rules of JCoBox\textsuperscript{m}.
This chapter provides a reference to the core API of JCoBox, i.e., the API that is mainly used by JCoBox users. The full API is available at http://softech.cs.uni-kl.de/~jcobox. The following interfaces and classes are given in this section:

- the `JCoBox` class (see Listing C.1),
- the `Fut` interface (see Listing C.2),
- future handler interfaces and adapter classes (see Listing C.3),
- the `Promise` interface (see Listing C.4).
package jcobox;

import java.util.concurrent.Callable;
import java.util.concurrent.TimeUnit;
import jcobox.annotation.Transfer;
import jcobox.internal.FutQ;
import jcobox.internal.JCoBoxCore;

public class JCoBox {
    public static void awaitTermination() { ... }
    public static void setAutomaticShutdown(boolean b) { ... }
    public static void shutdown() { ... }
    public static void exit(int status) { ... }
    public static void execute(Runnable r) { ... }
    public static <V> Fut<Void> runTask(Runnable r) { ... }
    public static <V> Fut<V> runTask(Callable<V> c) { ... }
    public static void sleep(long millis) { ... }
    public static void sleep(int duration, TimeUnit unit) { ... }
    public static void yield() { ... }
    public static void yield(long millis) { ... }
    public static void yield(int duration, TimeUnit unit) { ... }
    public static boolean isTransfer(Object o) { ... }
    public static void forceYield() { ... }
    public static <V> Promise<V> newPromise() { ... }
    public static <V> Fut<V> resolvedFuture(V v) { ... }
    public static <V> Fut<V> toFut(V v) { ... }
    public static void startBlocking() { ... }
    public static void endBlocking() { ... }
    public static void releaseCoBox() { ... }
    public static void acquireCoBox() { ... }
    public static <V> V boxify(Object target, Class<V> clazz) { ... }
}

Listing C.1: The JCoBox class API.
package jcobox;

import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public interface Fut<V> extends Future<V>, TransferObject {
    boolean isDone();
    boolean isResolved();
    boolean isCancelled();
    boolean hasException();
    Exception getException();
    V getValue();
    V await();
    V await(boolean forceSuspension);
    V await(long amount, TimeUnit unit) throws TimeoutException;
    V await(boolean forceSuspension, long amount, TimeUnit unit)
        throws TimeoutException;
    V get();
    V get(long amount, TimeUnit unit) throws TimeoutException;
    V getThrow() throws Exception;
    V getThrow(long amount, TimeUnit unit) throws TimeoutException, Exception;
    V awaitThrow() throws Exception;
    V awaitThrow(long amount, TimeUnit unit) throws TimeoutException, Exception;
    V awaitThrow(boolean forceSuspension, long amount, TimeUnit unit)
        throws TimeoutException, Exception;
    void suspendUntilAvailable();
    void suspendUntilAvailable(boolean forceSuspension);
    void suspendUntilAvailable(long amount, TimeUnit unit)
        throws TimeoutException;
    void suspendUntilAvailable(boolean forceSuspension, long amount,
        TimeUnit unit) throws TimeoutException;
    void blockUntilAvailable();
    void blockUntilAvailable(long amount, TimeUnit unit) throws TimeoutException;
    boolean cancel();
    boolean cancelWithInterrupt();
    <R> Fut<R> when(Handler<V, R> handler);
    Fut<Void> when(VoidHandler<V> handler);

    ...  // Fut.Handler, Fut.VoidHandler, Fut.Adapter, Fut.VoidAdapter, see Listing C.3
}

Listing C.2: The Fut interface.
public interface Fut<V> extends Future<V>, TransferObject {
    ...
    // methods see Listing C.2

    public interface Handler<V, R> {
        R ready(V value);
        R error(Throwable e);
    }

    public interface VoidHandler<V> {
        void ready(V value);
        void error(Throwable e);
    }

    public static class Adapter<V, R> extends CoObjectClass implements Handler<V, R> {
        public R ready(V value) { return null; }
        public R error(Throwable e) { return null; }
    }

    public static class VoidAdapter<V> extends CoObjectClass implements VoidHandler<V> {
        public void ready(V value) { }
        public void error(Throwable e) { }
    }
}

Listing C.3: The future handler interfaces and adapter classes.

package jcobox;

public interface Promise<V> extends StandardObject {
    public Fut<V> getFuture();
    public void resolve(V v);
    public void smash(Exception e);
    public void resolve(V v, Exception e);
    public boolean isResolved();
}

Listing C.4: The Promise interface.
This chapter presents plots for the different benchmark runs with increasing input parameters for the platforms Atom Client, Athlon, Core2Duo, and Opteron. The benchmarks on the Atom Server platform have only been measured for maximal input parameters, so they do not appear here. The plots for the Xeon platform are shown in Figure 5.14 on page 132.
Figure D.1: The pingpong benchmark executed on different platforms.

Figure D.2: The chameneos benchmark executed on different platforms.
Figure D.3: The ringmsgs benchmark executed on different platforms.

Figure D.4: The ringnodes benchmark executed on different platforms.
Bibliography


*Clojure website*, 2009.


Bibliography


Bibliography


[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.


[WCB01] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for
well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type
1994.


[WJH05] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for
Java. In Johnson and Gabriel [JG05], pages 439–453.

[Wor09a] Alexander Worret. Automated Product Derivation for the CoCoME
Software Product Line: From Feature Models to CoBoxes. Master's

[Wor09b] Peter Wortmann. Implementation of the CoBox Model in Scala. Master's

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-
oriented concurrent programming ABCL/1. In Norman Meyrowitz, 


[YT86] Yasuhiko Yokote and Mario Tokoro. The design and implementation of

[YT87] Yasuhiko Yokote and Mario Tokoro. Experience and evolution of concur-
Index

Symbols
\( \Gamma \), see Typing, variable
\( \Sigma \), see Typing, reference
\( \Theta \), see Type environment
\( \alpha \)-equivalence, 75 f
\( \alpha \)-renaming, 75
\( \vdash_d \), see Well-typed class declaration
\( \vdash_e \), see Typing, expression
\( \vdash_h \), see Well-typed method declaration
\( \vdash_p \), see Well-typed program
\( \vdash_t \), see Type, valid

Functions
\begin{align*}
& \text{copy}, 59 \\
& \text{farrefs}, 67 \\
& \text{fcopy}, 59 \\
& \text{foids}, 59 \\
& \text{freach}, 59 \\
& \text{futs}, 72 \\
& \text{inactive}, 72 \\
& \text{obj}, 67 \\
& \text{rdom}, 66 \\
& \text{refs}, 66 \\
& \text{resolvableFuts}, 72 \\
& \text{terminal}, 73
\end{align*}

Predicates
\begin{align*}
& \text{blockedtsk}, 71 \\
& \text{brokenexpr}, 71 \\
& \text{brokenresolve}, 71 \\
& \text{brokentsk}, 71 \\
& \text{fdaccess}, 70 \\
& \text{ffieldaccess}, 74 \\
& \text{futobj}, 66 \\
& \text{normalobj}, 66 \\
& \text{nullacc}, 70 \\
& \text{transferobj}, 66
\end{align*}

Notations
\( [e/x]e' \), see Substitution

A
ABCL, 36
ABCL/1, 36
ACID, 33
Act/1, 36, 42
Actor
body, 6
script, 6
Actor model, 6
AmbientTalk/2, 38
Amdahl’s Law, 1
Array, 92
ASP, 38
Asynchronous
\begin{align*}
& \text{call}, 83 \\
& \text{communication}, 19 \\
& \text{method call}, 7, 19
\end{align*}
Atomic block, 34
Atomicity, 33
Attribute, see Object, attribute
Axum, 40
Index

B
Blocked
task, 71
Body, see Actor, body
Boundary, see Box boundary
Box, 7
boundary, 8
Broken
equation, 71
resolve, 71
task, 70 f
Bytecode
rewriting, 128

C
C#, 41
CA, see Concurrent Aggregates
Call
asynchronous, 83
direct, 50
reentrant, 3
Claiming, see future claiming
Class, 3
cobox, 82
immutable, 86
transfer, 86
Clojure, 104
CoBox, 16
blocked, 23
class, 82
creation, 82
diagram, 135
inactive, 72
model, 13
Swing, 95
@CoBox, 82
CoCoME, see Common Component Modeling Example
Common Component Modeling Example, 153
Communicating event loops, 38
Communication
asynchronous, 19
synchronous, 24
Component
object-oriented, 7
runtime, 7
Concurrency
thread-based, 4
Concurrency model
data-centric, 35
Concurrent Aggregates, 40
Concurrent Smalltalk, 39
Configuration
initial, see JCoBox c, configuration, initial
terminal, 73
well-formed, 68
well-typed, 70
Configuration Equivalence, 78
Confluent, 29
Constructor, 90
@CoObject, 90
Cooperative
future claiming, 22
multi-tasking, 18, 84
scheduling, 18
Cooperative multi-tasking, 32
Coroutines, 32
Creol, 39
Critical section, 5

D
Data Race, 74
Data race, 27
high-level, 27
Determinism, 29
Diagram
cobox, 135
Direct
method call, 18
Direct call, 50

E
E programming language, 38
Eiffel\/, 36
Emerald, 35
Erlang, 36
Evaluation context, see JCoBox\textsuperscript{C}, evaluation contexts
Event-based programming, 32
Exception, 92
checked, 93
Expression
broken, 71
source, 54
External
object, 17

F
FairThreads, 33
Far
direct access, 71
object, 17
reference, 17
Field, see Attribute, 91
final, 91
Field Access, 74
Simultaneous, 74
Final
fields, 91
Fut<$V$>, 83, 189
Future, 22, 42, 83
claiming, 22, 42
cooperative, 22
exclusive, 22
explicit, 42
implicit, 42
resolvable, 73
resolved, 22
resolving, 22
transparent, 42
unresolved, 22

G
Graphical user interface, 95
Guards, 42
Guava, 34
GUI, see Graphical user interface

H
Hybrid, 37
near, 17
passive, 38
publishing, 8
reference, 3
service, 8, 17
state, 2
transfer, 25
Object-Oriented Component, 7

P
Parallel Object Monitor, 35
@PlainJava, 142
Point of sale system, 153
POM, see Parallel Object Monitor
POOL2, 36 f
seepoint of sale system, 153
ProActive, 104
Programming model
  multi-threaded, 2
  sequential, 3
Progress Lemma, 73
Promise, 24, 42 f, 84

R
Ready queue, 18
Reentrancy, 3
Reentrant
call, 3
  monitor, 5
Reference
  far, 17
  near, 17
  object, 3
Resume, see task, resume
Rewriting logic, 177
rules
  C-MODIFIEROk-CoBox, 53
  C-MODIFIEROk-Eq, 53
  C-MODIFIEROk-Obj, 53
  C-NoCycles, 53
  C-NoFieldHide, 53
  C-NoOverload, 53
  C-NoSelfExtend, 53
  C-OVERRIDEOk, 53
  C-UniqueNames, 53
  J-DIRECT-Call, 78
  J-FIELD-Select, 78
  J-FIELD-Update, 78
  J-Let, 78
  J-New-Local-Obj, 78
  R-AsyncCallFar, 62
  R-AsyncCallLocal, 62
  R-Congruence, 62
  R-DirectCall, 61
  R-FieldSelect, 61
  R-FieldUpdate, 61
  R-FutAwait, 61
  R-FutGet, 61
  R-FutResolve, 62
  R-Let, 61
  R-NewCoBox, 62
  R-NewObjFar, 62
  R-NewObjLocal, 61
  R-PromFut, 62
  R-PromNew, 62
  R-PromResolve, 62
  R-Task-Terminate, 61
  R-TaskResume, 61
  R-Yield, 61
  S-Def, 52
  S-Fut, 52
  S-Refl, 52
  S-Trans, 52
  T-AsyncCall, 55
  T-CoBox, 65
  T-Decl, 55
  T-DirectCall, 55
  T-Eps, 65
  T-FieldSelect, 55
  T-FieldUpdate, 55
  T-FutAwait, 55
  T-FutGet, 55
  T-Fut, 65
  T-Let, 55
  T-Method, 55
  T-NewIn, 55
  T-New, 55
T-Null, 55
T-Obj, 65
T-Program, 55
T-PromFut, 55
T-PromNew, 55
T-PromResolve, 55
T-Promise, 65
T-Ref, 65
T-Sub, 55
T-Tsk, 65
T-TypeCl, 55
T-TypeFut, 55
T-TypeProm, 55
T-Var, 55
T-Yield, 55
WF-CoBox, 69
WF-Configuration, 69
WF-PromInit, 69
WF-PromRes, 69
WF-SuspendSet, 69
WF-Transfer, 69
WF-UniqueCompIDs, 69
WF-UniqueObjIDs, 69

S
SALSA, 36 f
Scala Actors, 36
Scheduling
  cooperative, 18
Script, see Actor, script
SEDA, 41
Sequential Java Equivalence, 78
Sequential Object Monitor, 35
Sequential programming model, 3
Service, see object
Service object, 8
Sing#, 41
Singularity OS, 41
SOM, see Sequential Object Monitor
Stack frame, 3
State
  object, 2
Substitution, 50
Substitution lemma, 167
Subsumption, 54
Suspend, see task, suspend
Swing, 95
class
  example, 142
cobox, 95
@Swing, 142
Synchronous
call, 83
  communication, 24, 83
  method invocation, 3
Synchronous method call
  example, 3

T
Task, 18
  blocked, 71
  broken, 70 f
  creation, 18
  idle, 18
  ready, 18
  resume, 18
  suspend, 18
  suspended, 18
  yield, 18
Thorn, 40
Thread pool, 111
Thread-based concurrency, 4
Transactional memory, 33
Transactions, 33
Transfer
  class, 86
    example, 140
  object, 25
@Transfer, 86, 140
Type
  environment, 54
  valid, 53
Type soundness, 73
Typing
  expression, 53
  reference, 54
variable, 54

U
@Unchecked, 86

V
Variable
  free, 50

W
Wait-by-necessity, 42
Well-formed
  configuration, 68
Well-typed
  class declaration, 53
  method declaration, 53
  program, 53

Y
Yield, see task, yield
yield(), 84
Index