Merlin

A Simple Actor-Based Language

Department of Computer Science
Aalborg University
Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst
Phone +45 96 35 80 80
Fax +45 98 15 98 89
`http://www.cs.aau.dk/`

Title:
MERLIN: A Simple Actor-Based Language

Topic:
Programming Language Design, Concurrent Programming Models

Project Group:
d634a (room E4-117)

Members of Project Group:
Simon Kongshøj

Supervisor:
Thomas Vestdam

Semester:
DAT6

Project Period:
Feb 1st 2007 to Jun 15th 2007

Number of Copies: 4 (1 online)

Number of Pages: 108

Synopsis:

This project is primarily a study of the actor model of concurrent computation, structured around the design of MERLIN, a simple actor-based programming language intended primarily to explore aspects of actor-based language design.
The report gives an introduction to the basic concepts of concurrency, concurrent computation models, and the actor model in particular. An extensive documentation of the design process of MERLIN is given, with the main focus on analyzing the different language design options available within the actor-based programming paradigm. Each of the design decisions taken during the MERLIN design process is justified and put into perspective, by contrasting them with a selection of other actor languages. Finally, a simple abstract-syntax interpreter for the MERLIN language is developed and documented.
This work highlights some of the design issues of actor-based programming languages, including concurrency issues, language design and implementation challenges.

Preface

In the last decades, *concurrency* has become an increasingly important facet of the art of programming. While it sometimes seems that concurrency only adds more frustration to our programming lives, our programs ultimately exist in a universe that is characterized by a truly massive scale of concurrency. Whether we want to or not, we *have* to deal with concurrency.

However, many common programming languages are still, at their hearts, characterized by the assumption of a neat, orderly universe where everything happens one step at a time. In these languages, concurrency is regarded as the *exception*, rather than the rule. This report explores the language design space around a particular niche of concurrent programming theory, the *actor model*, in which the situation is quite the opposite: Programs in the actor-based view of programming consist entirely of communicating, concurrent entities. In this report, I show the basic theory behind concurrency and actors, and develop the simple, academic toy language Merlin as an example of how programming languages can be structured around actors.

Intended Readers

While an obvious readership for a master's thesis is the examiners who will be responsible for grading it, the main reason I chose to work on *this* particular subject was because I believe it is both important and intellectually fascinating. In this spirit, I have chosen to write the report for a somewhat wider audience, in the hope of providing inspiration to others. Thus, this thesis is targeted at the programming language design community as a whole.

Since one of my main points in this work is that we as programming language designers need to pay more attention to the concurrency abstractions we use in our languages, I have chosen to assume that the reader does not necessarily know a lot about concurrency. The second chapter of the thesis gives an introduction to concurrency, with a special focus on the actor model, working up from a very basic level. I do assume, however, that the reader can reason logically, knows the terminology, concepts and methods of programming language design, and has at his command a basic set of mathematical tools. These assumptions seem to be quite fair to make about programming language designers at large.

Acknowledgements

First and foremost, I'd like to thank my thesis supervisor, Thomas Vestdam, for his competent and useful advice throughout the process. While I suppose an argument could be made that thesis supervisors don't need acknowledgments since they get paid for what they do, I'm fairly convinced that Thomas spent more hours working with me than what is reflected in his paycheck.

Throughout the process of developing this thesis, I had many enlightening discussions about its finer points with some of my fellow students of computer science. I'd especially like to thank Arild Haugstad, Willard Rafnsson, Anders Terkelsen and Robert Olesen for providing me with a good surface to bounce ideas off of.

Finally, I'd like to thank Freja Grandjean for her patience and her very appreciated moral support throughout the whole process, even though I sometimes think the only thing that can possibly be more dangerous to one's sanity than being a computer scientist is dating one.

Contents

1. Introduction

The path to our destination is not always a straight one. We go down the wrong road, we get lost, we turn back. Maybe it doesn't matter which road we embark on. Maybe what matters is that we embark.
– Barbara Hall

Everything should be built top-down, except the first time.
– Alan J. Perlis (1922 - 1990)

This thesis is about the design of concurrent programming languages, and the programs written in them. It documents an exploration of a particular area of the concurrent programming language design space, through which the language Merlin is developed. Like other high-level languages, Merlin represents a conscious effort to make a complex phenomenon simpler to grasp. The most important tool used by any language to achieve this purpose is *abstraction*: Allowing complex, compound entities to be named and treated as simpler units. A controversial design decision in Merlin is the decision to discard the currently dominant abstraction for concurrency in programming languages: The thread.

1.1 Why Yet Another Programming Language?

It is almost a truism that concurrent programming is difficult. And yet, our programs exist in a universe that is characterized by concurrency at a massive scale, in which a mind-boggling number of events all take place at the same time. Humans evolved in a massively concurrent environment, and for the entire history of our species, an understanding of concurrent phenomena has spelled the difference between life and death. A deep-seated understanding of concurrency is hard-wired into our brains – which, in fact, are massively concurrent systems themselves. In this light, it seems almost unthinkable that we should be incapable of thinking clearly about concurrent phenomena in our computer programs. And yet, concurrent programming is notoriously error-prone (see the introduction of chapter 2 for a well-known and catastrophic example), and the difficulty of understanding concurrent programs is significant enough that many introductory programming textbooks leave concurrency out entirely, or treat it as an advanced topic.

A central theme of this thesis is the stance that a significant part of the well-documented difficulty of concurrent programming is a consequence of our concurrency tools, not the inherent complexity of concurrent phenomena. The fundamental concurrency abstraction used by most mainstream programming languages is the *thread*, a sequence of operations carried out in parallel with other threads, all accessing a shared memory. Threads do not resemble any natural phenomenon often observed by humans. The thread paradigm *does*, however, convey a fairly accurate image of how concurrent processing is carried out by a single-memory multiprocessor computer.

Thus, thread-based concurrency mechanisms are *low-level abstractions*. While threads may make a useful abstraction for low-level and medium-level programming languages, high-level application programming and scripting languages based on threads compromise their high-level nature when used to write concurrent programs: They provide high-level abstractions for sequential programs, but rely on low-level abstractions for concurrent ones. This is especially problematic for application programming and scripting languages, since the programs written in such languages are often centered around their interaction with the vastly concurrent world surrounding them.

There is a historical parallel: In the not too distant history of computer science, many high-level programming languages featured a memory abstraction which required manual management of heap-allocated memory. Pointers served as an abstraction over memory locations, and programmers manually allocated and freed heap memory. An entire class of programming errors represents faulty logic in dealing with manually-controlled heap memory: Buffer overflows, memory leaks and dangling pointers are examples. When the JAVA language was introduced in the early 1990s, the design choice to rely exclusively on automatic heap memory allocation was viewed as highly controversial[1]. Today, nearly all new high-level languages provide automatic heap memory allocation. It is not unthinkable that in the future, manual management of low-level concurrency mechanisms will be viewed in a similar light.

1.2 Problem and Method

If thread-based concurrency mechanisms are at the wrong level of abstraction for high-level languages, then what is the right level, and how can we get there? Over the years, several models of concurrency have been proposed as the answer to that question.

One such model is the actor model. Originally developed in the MIT artificial intelligence research community in the early 1970s, the actor model has the very appealing quality of *conceptual simplicity*: Its basic semantics are based on only two fundamental concepts, the *actor* and the *message*. From these, a full model of concurrent computation follows, complete with a set of "natural laws" for actor programs, allowing programmers to reason in a structured manner about their concurrent programs.

[1]Languages outside the programming mainstream had used automatic memory management for decades. The first language to rely on automatic memory management was LISP, developed in the 1950s.

1.2.1 Problem Statement

Is the actor model what we need for our concurrent programs? Certainly, the promises of conceptual simplicity and improved capacity for structured, methodical reasoning on programs are appealing qualities. However, it is also generally the case in computer science that there are no silver bullets, no perfect solutions to all our problems. It therefore seems appropriate to study the actor model, and its applicability to programming language design. This leads to the following problem statement:

CORE PROBLEM: *Which benefits and challenges does the actor model present as a basis for concurrent programming language design?*

1.2.2 Method Overview

In order to study this problem and propose answers to it, this thesis demonstrates the design and implementation of MERLIN, a small, academic "toy language" based on actor semantics. This method has two quite appealing qualities:

- An actual programming language, although very restricted in scope and not suitable for actual software development, will be developed. By documenting the design process of the language, and the challenges and issues that arise during this process, the computer science community as a whole will have access to the experiences gained in the process.
- By implementing the language, even with a very simple implementation, it is possible to get a good impression of which aspects of actor semantics can potentially pose implementation problems. Even if the implementation itself is not a production-quality language processor, a documentation of its development will highlight such issues.

Thus, the method used is *process-oriented*, rather than *product-oriented*: While the MERLIN language itself is certainly of some interest in its own right, it is not intended as the main contribution of this thesis. The main contribution is the documentation of the process that led to the MERLIN language.

The *ideal* method would be to develop a "real" language, give a high-quality implementation of it, and write a collection of non-trivial applications in it. However, programming language design is difficult and time-consuming work, and the time frame allotted for a master's thesis does not realistically allow adopting such an approach. The "toy language" method adopted in this thesis is an attempt to work from an *approximation* of this approach.

1.3 Contributions

This thesis documents a research project in the field of concurrent programming language design, focused on the design, implementation and application of MERLIN, a simple programming language based on the actor model of concurrent computation. The specific contributions given by this research can be summarized as follows:

- The fundamentals of concurrency theory have been investigated and presented. As part of this presentation, various aspects of different concurrency models are compared and contrasted. An introduction to the actor model of concurrent computation is given (see chapter 2).
- As part of the investigation of the actor model, the actor-based programming language MERLIN is developed. The design process is extensively documented, and the many design choices and tradeoffs are justified and put in a philosophical and technical context (see chapter 3).
- A simple, proof-of-concept MERLIN interpreter has been designed, and implemented as a SCHEME program. Although the interpreter is kept deliberately simple and unoptimized, it illustrates several of the technical challenges of implementing an actor language (see chapter 4).

1.4 Scope

The development of concurrent programming languages is an active research subject, with numerous interesting open problems. While there are many possible perspectives on the problems faced within the design and implementation of an actor language, it is unfortunately not realistic to do justice to all of them here. This ultimately means that several important and interesting topics within the field are only given a somewhat superficial treatment within this thesis:

Sequential programming : MERLIN is first and foremost a concurrent language. Only very little attention is paid to the sequential component of MERLIN, except for the situations in which decisions about the representation and capabilities of sequential subprograms directly affect the concurrent semantics of MERLIN.
RATIONALE: This project is primarily a study of concurrent programming language design. Although there are many interesting problems in sequential programming language design, addressing these would distract from the overall goals of this project.

Performance optimization : The MERLIN implementation is not fast. It is written as an interpreter running on top of another interpreter, which greatly limits its possible performance. Furthermore, the MERLIN interpreter does not attempt to use the most efficient algorithms in its implementation. The implementation is thus inefficient in both time and space.
RATIONALE: The implementation is designed first and foremost to illustrate the semantics of MERLIN, rather than being an efficient programming language implementation. Performance optimization could make the interpreter considerably less clear, and make it more difficult to understand its internals.

Garbage collection : The initial MERLIN implementation does not implement garbage collection beyond that of the implementation language, making it very memory-inefficient. This is obviously completely unacceptable in programming language implementations meant for "real-world" usage. Garbage collection of actor-based languages is known to be possible[Bak78].
RATIONALE: Developing a garbage collector for the MERLIN implementation could be quite time-consuming, and therefore require de-emphasizing other considerations that are more in tune with the general focus of the project.

Distributed programming : The actor model is very well-suited for distributed programming, since many of the concepts provided by the model are conceptually close to distributed programming concepts. In fact, the characteristic that actors can be concurrent in time implies that they can also be distributed in space. However, MERLIN does not implement any distributed programming facilities.
RATIONALE: This project primarily deals with concurrent programming, not distributed programming. A later implementation of MERLIN could include distributed programming facilities.

1.5 Related Work

The ERLANG programming language[AWV93], developed at the Ericsson Computer Science Laboratory, is a concurrent functional language designed specifically for developing highly reliable telecommunication applications. The entities responsible for concurrent computation in ERLANG are called *processes*, and can be viewed as recursive functions augmented with communication ports and message buffers. While ERLANG is not explicitly based on the actor model, its processes share many important characteristics with actors. It has several concepts in common with MERLIN: Both languages use a message-passing concurrency mechanism, neither language has any form of shared memory, and both languages emphasize abstraction mechanisms with roots in the higher-order functions of functional programming.

Another programming language which takes an actor-oriented view of concurrency is the embeddable scripting language Io [Dek05], developed by Steve Dekorte. The Io language is designed to fit in the same linguistic niche as the more well-known procedural language Lua: Both are small, embeddable high-level languages meant for application logic scripting. Io is a prototype-based object-oriented language following the tradition of the Self language, and includes an integrated actor-based concurrency mechanism. In Io, method calls can be either synchronous, future-based or asynchronous, and using either of the two latter possibilities yields actor-like concurrency semantics. Like Merlin, Io uses message-passing actors to implement concurrency, but takes a much more object-centered view of the actor concept.

Recently, the Scala language has adopted an actor-based concurrency mechanism, implemented as a library[HO07]. The Scala actor library was inspired by the Erlang concurrency model, and takes a similar basic form: Scala actors are lightweight sequential processes extended with a communication system. Scala extends the Erlang communication model by including, among other things, a mechanism for bidirectional asynchronous communication. Scala actors and their Merlin counterparts share the same fundamental characteristics: They exist in parallel, and interact using asynchronous message passing.

A similar project, bringing the actor model closer to the linguistic mainstream, is Salsa [VA01]: A library-based actor implementation, with many of the same characteristics as the Scala actor library, for the Java language.

2. Actor-Based Concurrency

All the world is a stage, all the men and women merely players.
– William Shakespeare (1564 - 1616)

The only thing that makes life possible is permanent, intolerable uncertainty; not knowing what comes next.
– Ursula K. LeGuin

Concurrent programming brings many benefits to computer science, and there are numerous reasons to study concurrency from a programming language design perspective. A frequently cited reason why the programming language design community should pay more attention to concurrency is the performance gains that can be reaped by exploiting the parallelism available in modern multi-core processor architectures. Although supercomputers have relied on massively parallel architectures for decades, multi-core processors are now beginning to enter the computing mainstream. The Intel Core and AMD Athlon processors are currently introducing multi-core processing into personal computers, and the IBM Cell processor featured in the current generation of Sony consumer gaming consoles is based on a multi-core architecture.

One method of improving the performance of programs on modern multi-core processors is by using automatically parallelizing compilers, some of which have been developed for several sequential programming languages. While it has been speculated[SL05] that such compilers are reaching the limits of extracting parallelism from programs in non-concurrent languages, they nevertheless show that it is possible to view parallelism as a *performance optimization* that can be optionally applied to a program.

A more compelling reason to study concurrency in programming language design is that a firm conceptual grasp on concurrency can increase our programming fluency: With concurrent programming languages, we gain the ability to express programs as systems of concurrent entities, rather than as strictly sequential processes. Some problem domains appear to lend themselves “naturally” to concurrent programming solutions[RH04], much like how recursive algorithms allow fluent and natural solutions of certain problems. Examples of application domains in which concurrency plays a critical role include network servers, embedded systems, simulations, video games and graphical user interfaces. These classes of programs necessarily involve concurrency, since they must interact with external entities that exist in parallel with the programs themselves.

With concurrency, we thus get another conceptual tool with which to design our programs. However, concurrent programming is notoriously difficult and prone to errors. A particularly infamous case of a concurrent programming error leading to disastrous consequences is the example of the Therac-25 computer-controlled radiation therapy machine[LT93]. In the late 1980s, this machine caused six known cases of massive radiation overdoses, resulting in severe injuries and deaths. An important cause of the Therac-25 accidents was improper synchronization between the operator interface and the equipment control – a directly concurrency-related error.

The Therac-25 control software was developed entirely in assembly language, and thus relied exclusively on low-level concurrency mechanisms. While assembly language is rarely used for applications programming today, the dominant concurrency mechanism in modern languages is still fundamentally a low-level abstraction, inspired by the nature of computing machinery rather than by principles of mathematics or by natural metaphors. Because of the complexity and awkwardness of low-level programming, it appears unlikely that the full potential of concurrent programming will be realized before the programming language design community adopts a higher level of abstraction for concurrency.

Chapter Overview

To develop concurrent programming languages, we must first understand the nature of concurrency itself. Therefore, the purpose of this chapter is to describe the theoretical foundation for the further development of the Merlin language. Since concurrent programming is generally not a particularly accessible subject, this chapter introduces all the terminology and theoretical fundamentals used throughout the rest of the report.

- *Overview of Concurrency Concepts*: This section gives an introduction to the basic concepts of concurrent programming. It introduces most of the concurrency terminology used throughout the rest of the thesis, and gives definitions of various important concepts. Furthermore, it describes some of the well-known issues and pitfalls that arise in concurrent programming.

- *Models of Concurrency*: This section investigates some of the basic concepts underlying the different models of concurrent computation that have been proposed. It gives a basic taxonomy of concurrent computation models, and describes the distinctions between deterministic and nondeterministic concurrency, shared- and distributed-memory concurrency, as well as various models for synchronization and communication in these models. This allows us to put later parts of the thesis into a wider context.

- *The Actor Model*: The final section of this chapter gives an introduction to the theory and history of the actor model, a nondeterministic, asynchronous, message-passing model of concurrent computation. Important concepts of the actor model are introduced and defined in this section.

2.1 Overview of Concurrency Concepts

We can naïvely define the concept of concurrency as the property that more than one computational process may be executing simultaneously. However, this definition dodges the issue, since it does not explain the exact meaning of the term "simultaneous". Let us therefore pause to consider the exact implications of simultaneous execution.

At a very fundamental level, we can view the execution of any computer program in terms of *executions*, which are ordered sets of *computational steps*. We can understand the nature of concurrency and sequentiality by investigating the orderings of the computational steps within executions. For this purpose, we consider two distinct orderings which can occur in executions:

Definition 1 (Total ordering) A relation $\rightarrow$ is a *total ordering* if and only if it satisfies the following properties:

- *Asymmetry*: If $a \rightarrow b$ and $b \rightarrow a$, then $a = b$.
- *Transitivity*: If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.
- *Completeness*: For all a and b, it must be the case that $a \rightarrow b$ or $b \rightarrow a$. □

Definition 2 (Partial ordering) A relation $\rightarrow$ is a *partial ordering* if and only if it satisfies the following properties:

- *Asymmetry*: If $a \rightarrow b$ and $b \rightarrow a$, then $a = b$.
- *Transitivity*: If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Thus, every total ordering is also a partial ordering. □

If an execution is totally ordered, then we say that it is *sequential.* Because of the completeness property of totally ordered sets, an ordering is defined between *every* computational step in such an execution. This means that there can be no "overlaps" in time between computational steps within a totally ordered execution. Therefore, the notion of a totally ordered execution corresponds to the intuitive notion of sequential program execution. A programming language is sequential if all executions of programs written in it are sequential.

Conversely, if the ordering of computational steps within an execution is a partial order, then the execution may contain steps between which no ordering is defined. We say that such steps are *concurrent*, since the lack of defined ordering between them implies that they may "overlap" in time. This corresponds to the intuitive notion of simultaneous execution. An execution is concurrent if it contains concurrent steps, and a programming language is concurrent if programs written in it can have concurrent executions.

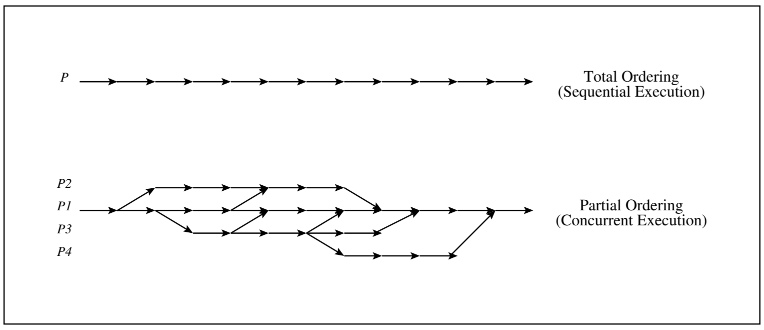


Figure 2.1: Total and partial orderings between computational steps

Individual sub-executions within a concurrent execution can be totally ordered. Without the ability to have non-overlapping steps, many important concepts, such as causality, could not be represented in a concurrent program[1]. We call a totally ordered sub-execution within a concurrent execution a *process*, not to be confused with the operating system concept of the same name.

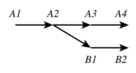
Both sequential and concurrent executions are assumed to be *causally ordered*: Causes must always precede effects.

2.1.1 Interleaving and Parallelism

What does it mean for two computational steps to overlap in time? By the above definitions, it simply means that there is no defined ordering between them. This implies that for two concurrent computational steps s_1 and s_2, both $s_1 \rightarrow s_2$ and $s_2 \rightarrow s_1$ constitute valid orderings. Therefore, concurrent processes may be *interleaved*, such that the execution has a single global sequence of computational steps, in which the actual computational steps each belong to independent, concurrent processes. If each computational step takes little physical time to complete, then this technique can give an illusion of simultaneous execution. Conceptually, this corresponds to the implementation of multitasking used in operating systems on single-processor computers.

If more than one processor is available, concurrent computational steps can be carried out in parallel. Thus, we can view parallelism as a *physical* phenomenon corresponding to the *logical* phenomenon of concurrency. In this view, parallelism is the implementation technique of exploiting logical concurrency in executions to improve the performance of programs. It is common practice in concurrency literature [RH04, MK99] to assume interleaving semantics, so that for any parallel execution, there exists at least one interleaving that is observationally equivalent to it.

[1]In fact, concurrency between computational steps arises when there is no causal relation between them[RH04].



A1, A2, A3, A4, B1, B2
A1, A2, A3, B1, A4, B2
A1, A2, A3, B1, B2, A4
A1, A2, B1, A3, A4, B2
A1, A2, B1, A3, B2, A4
A1, A2, B1, B2, A3, A4
A1, B1, A2, A3, A4, B2
A1, B1, A2, A3, B2, A4
A1, B1, A2, B2, A3, A4
A1, B1, B2, A2, A3, A4

Figure 2.2: A concurrent execution and its possible interleavings

The interleaving semantics of a concurrent system allows us to understand the phenomenon of *nondeterminism*, which is key to understanding why concurrent programming is difficult. Nondeterminism in concurrent executions occurs precisely because of the partially ordered nature of concurrent steps: Since there is no defined ordering between these, some ordering must be *chosen* in order to interleave them.

The problem of choosing orderings between concurrent steps is known as *scheduling*. When several processes execute in parallel, or when a scheduler is required to consider timing factors external to the execution, then various environmental interferences (such as other active executions) may cause the actual interleaving of computational steps to differ between individual executions of the same program. An example of this phenomenon arises when a concurrent execution relies on the operating system scheduler to interleave its processes: Since the operating system scheduler is also responsible for scheduling processes in other programs running on the system, its scheduling decisions can be affected by external factors.

2.1.2 Interaction

For concurrency to be useful for actual problem-solving, it is a necessary precondition that processes are capable of affecting each other – otherwise, having more than one of them would serve no purpose. We say that two processes *interact* when a computational step within one process causally precedes a computational step within another process. For example, if a process P_1 needs to access a resource which another process P_2 controls, then an access request by P_1 causally precedes the action P_2 takes in response to that request.

Race Conditions

We are now able to understand some of the fundamental problems that arise in concurrent programs, since these problems ultimately stem from the combination of nondeterminism and interaction.

Consider two concurrent processes $P_1 = P_2 =$ `{v := v + 21}`, and assume that the variable v, which they both access, is initialized to the value 0. Intuitively, an execution of P_1 and P_2 should terminate such that $v = 42$ – but does it? Let us investigate the situation by breaking down the definition of P_1 and P_2 down into primitive computational steps, as they might be realized on a physical computer:

1. Fetch the value v into a register r.
2. Add 21 to the contents of r.
3. Store the contents of r into v.

Since P_1 and P_2 are concurrent, their computational steps may be interleaved in any order. Unfortunately, some interleavings can lead to results that violate our intuition of correctness – and, usually, the intention behind a program. For example, consider the following interleaving of P_1 and P_2:

1. P_1 fetches $v = 0$ into register r_1.
2. P_2 fetches $v = 0$ into register r_2.
3. P_1 adds 21 to the contents of register $r_1 = 0$, yielding $r_1 = 21$.
4. P_1 stores 21 into v.
5. P_2 adds 21 to the contents of register $r_2 = 0$, yielding $r_2 = 21$.
6. P_2 stores 21 into v.

When this execution terminates, v is set to 21. A situation like this, where the correct outcome of an execution is dependent on a precise order of events, is referred to as a *race condition*. Program errors resulting from race conditions can be very difficult to track down, precisely because of the nondeterminism inherent in concurrent systems: Since the scheduler may choose different interleavings for each execution, a harmful result of a race condition may arise only under specific circumstances that can be difficult to reproduce. The Therac-25 malfunction mentioned in the introduction of this chapter was due to a race condition[LT93].

We say that an execution is *safe* if it is defined such that all possible interleavings of its processes yield correct results – in other words, if it is free from race conditions. To program reliable concurrent systems, we require a mechanism to enforce safety.

We can use our notion of implicit causal ordering to ensure the safety of processes. In the example above, the only correct interleavings of P_1 and P_2 are the two where one process completely precedes the other: $P_1 \rightarrow P_2$ and $P_2 \rightarrow P_1$. We require a mechanism which ensures that the completion of one process *causes* the beginning the other. Such a mechanism is called a *synchronization* between the two processes. A program segment is synchronized if it can only be executed by a specific number of processes at any one point in time. When one process executes a synchronized program segment, then the scheduler ensures that no other process may execute that segment until the first process has finished. We refer to entering and leaving synchronized program segments as *locking* and *unlocking* them, respectively.

Deadlock and Livelock

A synchronization mechanism is necessary to ensure safety, but also introduces new hazards. Synchronized resources allow a process to lock a resource, forcing other processes to wait until it is unlocked. Unfortunately, this allows processes to effectively monopolize resources by locking them indefinitely, preventing other processes from making any further progress. The classic *dining philosophers problem* (figure 2.1.2) [Dij71] illustrates this class of error.

Five philosophers are seated at a round table, sharing a delicious meal of instant ramen noodles. Each philosopher can be in one of two states, Eating his noodles and Thinking great thoughts. A philosopher requires two chopsticks to enter the Eating state, and will put his chopsticks back on the table upon re-entering the Thinking state. Unfortunately, a philosopher has no Working state, so the five philosophers can only afford a single chopstick each (this also accounts for their choice of meal). They have arranged the chopsticks so that there is one chopstick between each pair of philosophers. Assume that all hungry philosophers have the following, rather single-minded behaviour:

- When a philosopher feels hungry, he first picks up the chopstick to his right. If there is no chopstick to his right, he waits until his neighbour has dropped it.
- When a philosopher is holding a chopstick in his right hand, he picks up the chopstick to his left. If there is no chopstick to his left, he waits until his neighbour has dropped it.
- The philosopher then proceeds to eat, and drops the chopsticks when he is no longer hungry.

By this behaviour, a hungry philosopher will never put down a chopstick before he has eaten. If all five philosophers simultaneously pick up the chopsticks to their right, they will have picked up all chopsticks. In that case, no philosopher can pick up a left-hand chopstick, which in turn means that no philosopher will ever eat, and no philosopher will ever put down a chopstick.

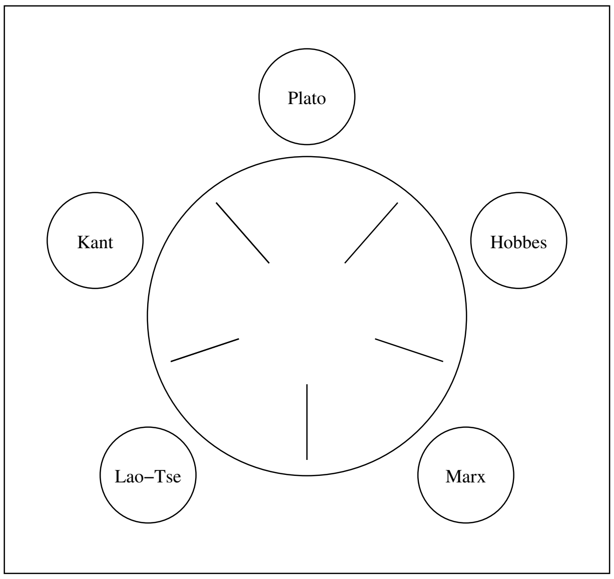


Figure 2.3: Dining Philosophers

We refer to this situation, where each involved process is waiting for the others to finish, while also preventing the other processes from finishing, as a *deadlock*. There are four necessary and sufficient conditions for deadlocks to arise[Eng04]:

No Preemption : Once a process has locked a synchronized resource, it cannot be preempted by another process. The process must voluntarily release the resource. In the dining philosophers example, this condition is satisfied because once a philosopher has taken a chopstick, he will hold on to it until he has eaten. If it was possible for another philosopher to steal a chopstick from his neighbour, then an unscrupulous philosopher will eventually finish eating and put down his chopsticks, breaking the deadlock.

Circular Waiting : The processes involved in the deadlock must form a cycle. This means that ultimately, deadlock is a “chicken-and-egg” paradox, in which any process in the set is dependent on its own completion before it can complete. In the example, this condition is satisfied because the philosophers are seated around a circular table with only five chopsticks. If the philosophers were seated in a row, with an extra chopstick at one side, then it is guaranteed that two chopsticks will always be available for a philosopher seated at one of the far ends, thus breaking the deadlock.

Hold-and-Wait : Once a process in the deadlock has locked a resource, it will not release the resource before it has completed its task. In the example, this condition is satisfied by the definition of a hungry philosopher's behaviour: He will not put down a chopstick before he has eaten. It would be possible to break the deadlock if a philosopher could put down a chopstick when he realizes that the other chopstick is not available.

Mutual Exclusion : Only one process may keep a lock on a resource at any given instant. In the example, this condition is satisfied because two philosophers cannot eat with the same chopstick simultaneously. If the philosophers know a technique allowing two people to share a chopstick in use, then the deadlock could be broken.

A related problem is *livelock*, in which a set of processes are active, but in such a way that the activity of each process prevents the other processes from making progress. A commonly-used metaphor for livelock is the "hallway dance": Two people meet in a narrow hallway, and to avoid an unpleasant collision, both people step aside to allow the other to pass. When both keep stepping to the same side at the same time, they are livelocked. Livelock is generally not as severe a problem as deadlock, since even small differences in timing between the processes will eventually break a livelock.

Unfortunately, it is the mechanism that prevents race conditions that also introduces the possibility of deadlock and livelock. This means that deadlock and livelock can only be universally eliminated by also eliminating synchronization – and in that case, it is impossible to guard against race conditions. These, in turn, are made possible because of the inherent nondeterminism of concurrent processes. Thus, concurrent programming will frequently require the programmer to consciously balance considerations of *safety* (freedom from race conditions) with considerations of *liveness* (freedom from deadlock).

Therefore, practical concurrent programming languages are *necessarily* subject to some complications that do not arise in sequential programming languages. This makes it *inherently* more difficult to write correct concurrent programs: A concurrent program must be correct for *all* possible interleavings of its computational steps, whereas a sequential program by definition only has one possible interleaving. To cope with this additional difficulty, it is useful to introduce constraints on the set of possible interleavings, bringing down the total number of possible interleavings and making it easier for programmers to understand the concurrent execution of their programs.

Such constraints can be introduced into a concurrent language by choosing a particular set of concurrency abstractions, ideally a set which allows a programmer to easily understand and conceptualize the interactions and synchronizations taking place. In high-level languages, this is done by basing the concurrent semantics of the language on a specific *concurrency model*, allowing the programmer to understand the interleaving possibilities in terms of higher-level abstractions. Because concurrency models are such an important component of concurrent programming language design, we turn our attention to the design of concurrency models.

2.2 Models of Concurrency

A *model of computation* is an abstract, formal system which defines a programming language and its semantics. By definition, a model is a *representation* of some object being modelled. A computation model is thus an abstract representation of computation, reather than a concrete computation mechanism.

Computation models are useful to practicing programmers precisely because they allow a programmer to disregard the finer details of how computations are realized. The mechanism used to accomplish this is by providing a set of abstractions, allowing the programmer to make some simplifying assumptions about the programs written within a particular model. For example, a programmer working with a model which supports recursion can assume the existence of a call stack to handle the parameters and return values of each level in a recursive procedure, and a programmer working within a pure functional model can assume that no program element has side effects. It does not matter if the concrete realization of recursion uses a continuation chain rather than a stack, or if the implementation of a functional language uses assignment statements "behind the scenes", as long as there is no *observable* difference between the model and its realization.

A *concurrency model* is thus a computation model which provides a set of abstractions about the nature of concurrent computation. Models of concurrency can help reduce the difficulty of writing and understanding concurrent programs, just like the aforementioned examples showed how sequential computation models can ease the understanding of sequential programs. We can understand concurrency models in terms of their answers to three fundamental questions:

- Does the model have *observable nondeterminism*?
- Which *memory model* does the model use?
- Which *synchronization model* does the model use?

The answers to these three questions have profound implications on the basic anatomy of the languages based on them, and the programs written in those languages. The universe of concurrency models can be understood as a hierarchical taxonomy, divided on the lines of these three, fundamental questions (see figure 2.2). The rest of this treatment of concurrency models will be based around this taxonomy, and the following discussion should make the structure of the taxonomy clear.

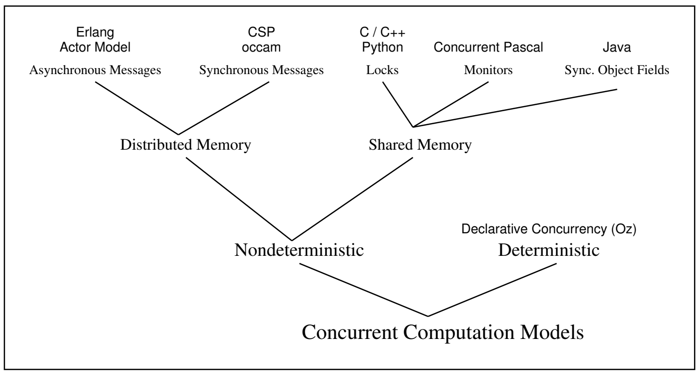


Figure 2.4: Taxonomy of Concurrent Computation Models

2.2.1 Deterministic Concurrency Models

Arguably the most important characteristic of a concurrency model is whether or not it is observationally deterministic. Observationally deterministic models have some very compelling advantages over nondeterministic models, but are also subject to severe limitations.

A concurrency model is *observationally deterministic* if any nondeterminism that arises in a concurrent execution is not observable by the programmer. This property has profound implications for programs written in an observationally deterministic concurrent language, an important one being that *race conditions are impossible* in such a language. This follows trivially from the definition of race conditions: Race conditions arise when the nondeterministic ordering of computational steps in an execution can lead to observably different results.

Observationally deterministic concurrency can be modelled with processes operating on *dataflow variables*[RH04]. A dataflow variable is a data abstraction which has the property of implicit, data-driven synchronization, and which can be in either a bound or unbound state. Here is an informal semantics of dataflow variables:

- When a process attempts to read a bound dataflow variable, it obtains the value and proceeds normally.
- When a process attempts to read an unbound dataflow variable, it silently blocks until the variable becomes bound.
- Assigning a value to an unbound dataflow variable causes it to become bound, which implicitly reactivates all processes blocked on it.
- Attempting to assign a value to a bound dataflow variable is an error.

Programming languages based on multiple processes operating on such variables have observationally deterministic concurrency because of the single-assignment semantics of dataflow variables, and because of the silent, implicit blocking. Due to the combination of these two factors, assigning a value to a variable causally precedes reading it, which imposes an ordering on the computational steps in the execution of a program in this model. This ordering has the property that only the *intermediate* steps between assignments and reads are subject to nondeterminism, the sequence of assignments and reads will be identical for each execution of the program, given the same input.

It is important to note that because of the blocking semantics of dataflow variables, observationally deterministic concurrent programs can enter deadlocked states. Deadlocks do not affect the observational determinism of a program: If a given set of inputs to an observationally deterministic program leads an execution to a deadlocked state, then that set of inputs will lead *all* executions of the program to a deadlocked state. This makes deadlocks considerably easier to debug in observationally deterministic programs. This model is presented in *Concepts, Techniques and Models of Computer Programming*[RH04], in which it is implemented as a subset of the Oz language.

However, many application domains are *naturally* characterized by observable nondeterminism, and this appears to be especially true for application domains in which concurrency plays an important role. For example, consider a client-server application, in which a server process maintains ongoing communications with a set of client processes. If the client processes are independent (for example, if they are running on different physical machines), then they can send commands to the server process in an unpredictable order. However, this unpredictable order is in fact an instance of observable nondeterminism! Unless the server can precisely predict the exact sequence of incoming commands, then such an application is impossible to model without introducing observable nondeterminism[RH04].

This greatly limits the relative expressive power of observationally deterministic concurrency models, and explains why the determinism question is placed at the root of our taxonomy: Programs exist that can be modelled in nondeterministic concurrency models, but not in deterministic ones[RH04]. Conversely, as we shall see, nondeterministic concurrency models are equal in expressive power: It is possible to implement one of those models in terms of another.

2.2.2 Shared-Memory Concurrency Models

In *shared-memory* concurrency, programs are represented as a single, shared set of memory cells, which a number of concurrent processes operate upon. Any process may read and modify the shared memory. Conceptually, processes in a shared state model are *active* entities, which may make decisions and perform activities based on both their local concerns and their observations of the store. The store itself is entirely passive, and only changes when some process modifies it. Because a process can only affect another

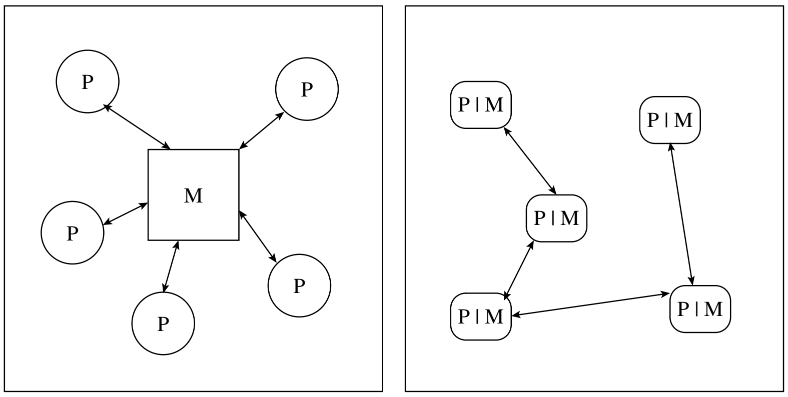


Figure 2.5: Shared (left) vs. Distributed Memory (right) Concurrency Models

process by modifying the shared store, the interaction between processes in this model is *indirect*.

In part because the shared state model closely mimics the actual operation of a single-memory multi-processor computer, it is often used to implement concurrent, "multi-tasking" processes in operating systems [Eng04, Lov05]. Note that in operating system terminology, a *process* denotes an independent execution which has its own memory space, whereas an operating system *thread* more closely resembles the usage of the word "process" adopted in this chapter. Threads in operating systems are "lightweight processes", which share the memory space of all other threads in the program that spawned them. Different operating systems provide different thread APIs, with the two in most widespread use being POSIX pthreads and the Windows thread API. Since most shared-memory concurrency implementations are based on operating system concepts, and thus adopt a similar terminology, we will refer to processes within shared-memory concurrent models as "threads".

The shared-memory model is very common in programming practice. The two aforementioned thread APIs are often used to develop concurrent programs in C and C++. Furthermore, the shared-memory model is the default mechanism of concurrency in both JAVA and C#, and is also used in the scripting languages PYTHON, PERL and RUBY, among others. A shared-memory concurrency model for the SCHEME language is given in SRFI-18.

The key synchronization problem in shared-memory concurrency models is maintaining the consistency of the memory while multiple, unpredictable processes are accessing it. Several mechanisms for this have been proposed, including (among others) lock variables, monitors and synchronized object fields.

Lock Variables

The most common mechanism for coordinating access to the shared store in shared-memory concurrency models is by using various forms of *lock variables.* The simplest lock variable is the *mutex*, which ensures *mutual exclusion.* A mutex can be in the states Locked or Unlocked. The informal semantics of a mutex can be summarized as follows:

- If a mutex is Unlocked, and a thread attempts to lock it, then it becomes Locked.
- If a mutex is Locked, and a thread attempts to lock it, then the attempt to lock it blocks, preventing the thread from progressing.
- If a mutex is Locked, and the thread holding the lock unlocks it, then it becomes Unlocked, unless any threads are blocking on lock attempts. In that case, one of them is reactivated, and the mutex remains Locked.

Mutexes can be used by threads to guard against race conditions by locking a mutex before entering a potentially unsafe state, and unlocking it again upon leaving that state. This guarantees that at most one thread will ever be in that state at any point in time. A somewhat more flexible generalization of the mutex concept is the *semaphore*, which is based on an integer counter rather than a binary Locked/Unlocked state.

Unfortunately, lock variables are very prone to deadlock. When any thread requires multiple locks, then *all* threads that require those locks must lock and unlock them in the same order, or a deadlock will result[Eng04]. This has several unfortunate implications for programming with lock variables. One is that lock-based subprograms are *not composable*: It is not possible to combine two correct lock-based subprograms and assume that the result is also correct. Another is that locks cause an *abstraction leak*: Programmers who use a lock-based procedure must be aware of the order in which that procedure acquires the locks, if another part of their program requires those locks as well.

Monitors

Monitors[Hoa74] represent an attempt to combine shared-memory concurrency with structured programming methods. A monitor consists of a set of protected memory cells, as well as a set of procedures which can operate on those cells. Like a module in some programming languages, a monitor *exports* a set of procedures for use by other parts of the program, and enables synchronization by keeping an implicit mutex lock. Whenever one of the exported procedures is called by the outside world, the mutex is implicitly locked, and when the procedure finishes, the mutex is implicitly unlocked. This guarantees mutual exclusion between the exported procedures of a monitor.

As is evident from this description, it is possible (and quite straightforward) to implement monitors in terms of lower-level lock variables. The reason for linguistic support for monitors follows the general rationale of the structured programming movement: Like structured sequential programming constructs leave less room for programmer error than unstructured ones (such as the infamous `GOTO` statement[Dij68]), so too do monitors leave less room open for error than explicit mutexes or semaphores.

Monitors enjoyed some popularity in the concurrent language design community in the late 1970s and early 1980s, and some notable examples of monitor-based languages from this era include Concurrent Pascal and Mesa. Since then, there has been a general trend in the language design community to move away from monitors and back to explicit lock variables[Rep92], because monitors mix the otherwise separate concerns of modularity and synchronization.

Synchronized Object Fields

In object-oriented languages, it is possible to use a concept similar to that of monitors to achieve synchronized access to the fields of an object. In this model, every object keeps an internal mutex variable, which is implicitly locked when a method is called, and unlocked when the call returns.

An important difference between monitors and synchronized object fields is that the object in this model may have publicly accessible methods that are *not* synchronized – in other words, the programmer must explicitly choose which of the methods of an object should be synchronized and which shouldn't. The act of balancing liveness and safety in such languages is largely a matter of deciding which elements of the public interface of an object should be synchronized. Like monitors, these mechanisms can be (and usually are) implemented in terms of low-level lock variables, but linguistic support for them is meant to reduce the likelihood of programmer error.

The JAVA language is a notable example of a language which provides synchronized object fields[Eck98], as well as low-level lock mechanisms.

2.2.3 Distributed-Memory Concurrency Models

In contrast to the shared-memory approach, a *distributed-memory* concurrency model relies on completely independent processes, which have no shared memory between them. In such a model, every process has its own local memory, which is not directly visible to any other process. This implies that *all* data in the system is managed by processes. Instead of interacting indirectly through a shared store, processes in such models interact by *communication*, which is why this model is often known as *message-passing concurrency*[2]. Unlike threads in a shared-memory concurrency model, message-passing processes are *reactive* entities, which only perform activities in response to stimuli.

As is implied by the name, distributed-memory concurrency models are especially well-suited for distributed systems, since a distributed system is defined precisely as a

[2]Throughout the rest of this thesis, the terms "distributed-memory concurrency" and "message-passing concurrency" are used interchangeably.

system composed of independent entities which interact only using messages[CDK94]. Therefore, many of the theoretical results of the distributed programming field apply to any purely message-passing concurrent system, regardless of whether it is running on a single physical machine or if it is distributed in space across multiple machines.

However, distributed-memory concurrency is relatively rare in programming practice, and is typically provided by languages that are designed explicitly for concurrent or distributed programming. For example, ERLANG, E and OCCAM provide message-passing concurrency primitives. In the scripting language community, distributed-memory concurrency is also comparatively rare, but some examples do exist. Notably, the IO language uses a purely message-based concurrency model, and the Stackless dialect of PYTHON extends stock PYTHON with communication channels and "lightweight threads". Many theoretical models of concurrency are based on communicating, distributed processes; examples include CSP, the π-calculus and the actor model.

Synchronization in distributed-memory models can be understood in terms of the synchronization semantics of the communication primitives in the model. In nearly all cases, the message reception primitive blocks, although a few message-passing concurrent languages provide *timeouts* or *polling* reception primitives. However, the synchronization semantics of sending primitives is a central choice in the development of a message-passing concurrency model. We will consider both synchronous and asynchronous message sending primitives, assuming blocking reception primitives for both.

Synchronous Communication

A *synchronous* message send will block the sending process until the receiver is ready to process the message. In this case, the message sending operation functions as an implicit synchronization mechanism. Formal concurrency models based on synchronous communication include CSP and the π-calculus, and concurrent programming languages which implement synchronous communication models include OCCAM and PICT.

Informally, the semantics of synchronous messaging can be described as follows:

- Send m, P: Send the message m to target process P. This call blocks until P issues a corresponding Receive.
- v = Receive: Receive a message from the communication port and bind the contents to the name v. If no message is available, this call blocks until another process Sends a message to this process.

Recall that all data objects in a message-passing system are kept in the local stores of processes. This means that for any other process to access data kept in the store of another process, it has to explicitly request the data by sending a request message to that process. Message-passing programs can use synchronous messages to ensure against race conditions by simply having each process only accept incoming messages when it is in a safe and consistent state.

This has a considerable benefit over lock variables: It does not suffer from leaky abstraction. A programmer can write processes that send synchronous messages to other processes without any knowledge of the internals of the receiving process, except that it will eventually Receive the sent message.

However, synchronous messaging suffers from two problems: *Excessive blocking* and *recursion deadlock*. Both problems stem from the semantics of Send. First, when a process Sends a message to another process, it is unable to do any more work before the receiving process issues the corresponding Receive. This is a parallelism bottleneck, and considerably lowers the degree of concurrency in a program. Secondly and more importantly, *any* recursive message pattern leads to an instant deadlock. This follows from the semantics of Send: When a process Sends a message to itself, it will block and wait for the corresponding Receive, and therefore, it never proceeds to call Receive.

Asynchronous Communication

An alternative communication mechanism for message-passing concurrency models is *asynchronous messaging*. Asynchronous messaging assumes the existence of message buffers in processes, and most concurrency models assume that message buffers are of unbounded size. Message buffers may be queues with strictly First-In-First-Out (FIFO) semantics, or they may use a more sophisticated selection algorithm to choose which of the messages in the buffer to process next. Formal concurrency models based on asynchronous messaging include the Asynchronous π-calculus, as well as the actor model (see section 2.3). Practical programming languages which implement asynchronous message-passing concurrency include ERLANG, SCALA and IO.

As with the preceding examples, we give informal semantics of asynchronous messaging:

- Send m, P: Place the message m in the message buffer of the process P.
- v = Receive: Receive a message from the message buffer and bind the contents to the name v. If the buffer is empty, this call blocks until a message arrives in the buffer.

Observe that in the asynchronous model, only the Receive operation blocks. This has major implications for programs in the model. Notably, both the excessive blocking and the recursion deadlock problems of the synchronous model no longer exist. The parallelism bottleneck is eliminated because a sending process may continue working after issuing a send. This increases the total concurrency in the system, at the cost of message buffer overhead. Recursion deadlock is eliminated because a process can place a message in its own buffer and immediately move on, thus eventually reaching the next Receive.

Unfortunately, asynchronous messaging introduces the risk that a process can propagate *stale information* to other processes, which can lead to incorrect results, similar to those that arise from race conditions in shared-memory concurrent systems. Consider a process P_1 which changes its internal state (say, the value of the variable v) upon receiving a message m, but before it can change state, it must consult another process, P_2. It Sends a message to P_2 and then calls Receive, awaiting the response from P_2. What happens if a third process, P_3, requests the value of v? Then P_1 will reply with a value of v which will be changed very soon afterwards, and the system enters an inconsistent state.

This problem can be alleviated using *insensitivity*. Insensitivity means that after sending a request message, a message-passing process can partially disable message reception so that the only message it will receive is the reply. This prevents it from propagating stale information, but unfortunately, it reintroduces the problems inherent to synchronous messaging – essentially, insensitivity is a mechanism which allows asynchronous communication systems to simulate synchronous ones.

2.2.4 Evalution of Concurrency Models

The core problem of this thesis stated that the purpose of this work is a study of a particular concurrency model, the *actor model*. Having investigated some of the important properties of concurrency models, we are now in a position to describe exactly *why* the actor model is interesting to study, compared to all the other concurrency models.

Which model of concurrency is "right" for a programming language is a controversial question, and while there is no academic consensus about its answer, it is important to consider the question in context. Before choosing a concurrency model for a language, it is thus important to briefly reflect on the purpose of that language. While a full discussion of programming language requirements is postponed to section 3.1.2, we already touched upon the purpose of studying concurrency in the introduction of this chapter. We motivated the study of concurrency was motivated with the following factors:

- Better utilization of parallelism.
- Improving programming fluency.
- The "natural" need for concurrency in certain problem and application domains.
- Using higher-level programming abstractions to reduce programming errors.

We will concentrate on models suitable for a *general-purpose language*, useful to conceptualize concurrent solutions to many different problems. Because we have observed that concurrent programming is inherently more complex than sequential programming, this language should aim for *simplicity*, to avoid introducing any more complexity into

	Shared	**Distributed**
Memory	Single global store	Multiple local stores
Processes	Active	Reactive
Process interaction	Indirect	Direct
Synchronization	Memory locking	Message synchronization

Table 2.1: The shared and distributed memory models

an already complex subject. Finally, the language should have a *high abstraction level*, because of the observation that the currently dominant paradigm for working with concurrency works at a too low level.

Deterministic vs. Nondeterministic Concurrency

We have seen that while deterministic concurrency models have many desirable properties, they are also strictly less powerful than nondeterministic ones: There are programs that can be expressed in nondeterministic models that are impossible to express in deterministic ones, precisely because certain programs *require* the ability to express observable nondeterminism due to their problem or application domain. This indicates that a language based upon an observationally deterministic concurrency model cannot be truly general-purpose.

It is still possible to make some use of deterministic concurrency in a general-purpose language: Both the E and Oz languages include deterministic *subsets*. These languages are based on *layered* language designs[Roy06], in which several concurrency models are available, and where the programmer can choose between deterministic and nondeterministic concurrency depending on context and need. However, a layering of multiple models increases the overall complexity of the language, since the language has to include several linguistic concepts for accomplishing the same task. This conflicts with our goal of simplicity. We therefore conclude that a *nondeterministic* model is most suitable.

Shared vs. Distributed Memory

Having chosen a nondeterministic model, we now have to consider whether the language should be based on a shared or distributed memory model. Table 2.2.4 reviews the identifying properties of both models.

It is important to note that the two memory models are equivalent in expressive power. This follows from the fact that a message-passing communication system can be developed in terms of shared memory primitives, and vice versa. For example, the Scala message-passing concurrency system is implemented in terms of the shared memory model provided by the Java Virtual Machine[HO07], and the paper *Actors and Continu-*

ous Functionals[HB77] gives an implementation of shared memory cells as message-based, distributed-memory actors.

Despite this equivalence, the different abstractions provided by the two models give them distinct sets of strengths and weaknesses, both because of implementation factors and because of the quite different *styles* of programming they encourage. This means that one model may conceptually "fit" a particular problem domain better than the other. We have already noted that the message-passing model lends itself well to distributed programming applications, and that the shared-memory model is well-suited for giving a conceptual model of operating systems on single-memory computers. We can contrast some of the relative strengths and weaknesses of the two models relative to the motivations of concurrency we identified in the introduction.

We consider the *implementation efficiency* (and the related *memory usage*) of the models, which is important for the goal of improving program efficiency using parallelism. We consider the *abstraction level*, which is important for the goal of programming fluency. Finally, we consider the facilities for *reasoning* about programs, which is important for understanding their correctness.

Efficiency : Shared-memory concurrency is often implemented directly in terms of low-level operating system primitives. This means that shared-memory concurrency implementations can often be implemented with very low overhead, and the concurrency primitives of shared-memory concurrency implementations can thus be very fast. Message-passing concurrency implementations are often implemented as compound procedures abstracting over shared-memory primitives, reducing the efficiency of message-passing primitives.

Memory Usage : A shared-memory concurrency implementation can be very light on memory resources, precisely because of the memory sharing. Each process needs only store pointers or references to the shared state, and shared data values do not need to be copied. The lack of copying further improves the speed of the model. In contrast, message-passing concurrency implementations must perform copies of data values when these are sent in messages.

Abstraction Level : The shared-memory model has been criticized[SL05] for having a too low level of abstraction, since it relies on a set of abstractions that are very machine-oriented in nature. This means that shared-memory concurrent programs often suffer from excessive clutter of low-level concurrency management code. In contrast, message-passing models are often founded in mathematics[Mil75], or in abstractions designed to resemble physically observable phenomena[HB77].

Reasoning : Causal relations in message-passing systems have been studied intensely, and are reasonably well-understood[HB77]. For example, it is necessarily the case that sending a message causally precedes the reception of that message. Such "natural laws" for message-passing systems allows reasoning on them by reasoning on causality chains. There are no similar reasoning methods for shared-memory concurrency models[RH04]. This makes message-passing systems simpler to reason about than shared-memory systems.

We can make some observations based on the above. When processor and memory efficiency are of high importance, *shared-memory* concurrency is most applicable. When it is important to conceptualize programs at a high level of abstraction, or to reason about the correctness of programs, then *message-passing* concurrency is most suitable. The introduction identified the goal of improving programming fluency as more important for this project than that of improving program performance, so we choose *message-passing* concurrency for our concurrency model.

Synchronous vs. Asynchronous Communication

Having identified that our model will be based on distributed memory and message-passing, the final question to address is whether we should use synchronous or asynchronous communication. This is a hotly debated topic in the message-passing concurrency community, but it is important to note that just like with the memory models, the two models are in fact equivalent in formal expressive power. As discussed in section 2.2.3, an asynchronous system can model a synchronous one by using a combination of acknowledgement messages and insensitivity. Likewise, a synchronous system can model an asynchronous one by using a synchronous process to model message buffers, and make sure that such buffer processes will always enter a state where they will Receive the next message[Rep92].

However, synchronization is rarely an "all or nothing" consideration. Some activities must be synchronized, whereas others should be performed asynchronously[3]. The synchronous model performs synchronizations implicitly, but has the drawback that the programmer must resort to modelling a message buffer when he *doesn't* want a synchronization. Likewise, the asynchronous model requires the programmer to model synchronicity using insensitivity when he *does* want a synchronization. This means that an important stylistic difference between programs in the two models is whether or not controlling the degree of synchronization is viewed *subtractively* or *additively.* It appears at this point that the mechanism for simulating asynchronous semantics within a synchronous model is more cumbersome than vice versa, so we choose *asynchronous communication* for our model.

In summary, we have found that what we require is a *non-deterministic, asynchronous message-passing* model. One model that meets these criteria is the actor model, and thus, we now turn our attention to a deeper study of that model.

[3]This corresponds somewhat to the balancing act between safety and liveness.

2.3 The Actor Model

The *actor model* is a concurrent computation model, originally developed in the 1970s for artificial intelligence applications. Apart from its early artificial intelligence roots, the actor model has been used as a tool to explore programming issues within distributed, concurrent and parallel systems. The universal unit of computation in the model, from which it gets its name, is the *actor*. While individual actors are simple entities with a very limited repertoire of actions, a community of actors can perform sophisticated computational tasks through cooperation and coordination. The actor model is a non-deterministic, purely message-passing model based on buffered, asynchronous messages.

2.3.1 Basic Concepts

The core idea of the actor model is the philosophy of representing programs as societies of actors, which coordinate their actions using a pure asynchronous message-passing communication system. Actor-based systems have several important, defining characteristics:

Locality : An actor system has no concept of global state. In other words, there are no data values that exist independently of the actors that make up the system.

Opacity : Individual actors can have local state, but this state is completely concealed from the outside world. There is no way for any actor to directly read or modify the state of another actor; actors must transmit data to one another in messages.

Independence : Actors are concurrent entities which exist independently of each other. Thus, they may carry out actions in parallel, and can exist distributed on several physical computers.

The actor model unifies some of the concepts from the object-oriented and functional programming paradigms. Like objects in object-oriented programming, actors are independent entities with identity that encapsulate an internal state, and like functions in functional programming, actor computation does not use destructive memory update[HA86].

Actors

The fundamental primitive of the actor model is the actor. An actor is a reactive object which consists of an *address* and a *behaviour*. The address is a globally unique identifier, and serves as the actor's identity. Addresses in the actor model are first-class values, and may be freely communicated to other actors[Agh86].

A behaviour, in turn, consists of a set of *acquaintances*, which are addresses of other known actors, as well as a *script*, which is a piece of code the actor executes when

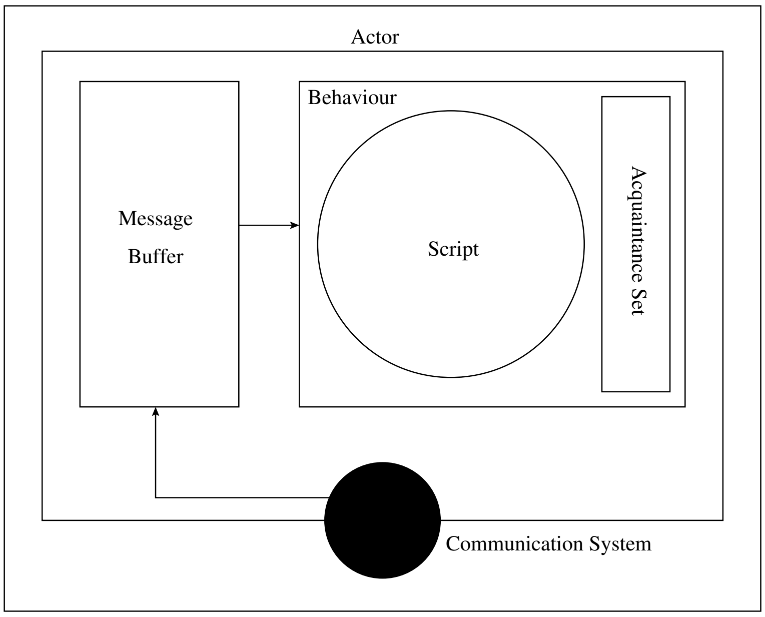


Figure 2.6: Internal Anatomy of an Actor

it receives a message. The acquaintance set is fundamental to how the actor model represents state: Rather than having a set of local, assignable variables, an actor has a set of other actors it can communicate with. The members of this set may be completely unknown to other actors in the system. Since actors are reactive objects, their behaviour scripts are only ever invoked in response to received messages. Scripts are combinations composed of the following *primitive actions*[Man88]:

Conditional Decision : The actor may make simple conditional decisions, based only on information in its acquaintance set, or the message it received.

Message Transmission : The actor may send a finite number of messages to actors that are either in its acquaintance set, or referred to in the message it received.

Actor Creation : The actor may create a finite number of new actors. The addresses of these actors may be communicated to other actors.

Behaviour Replacement : The actor *must* specify a replacement behaviour, which is then used when it receives the next message. This is how the actor model represents state: The set of local values of an actor *is* its acquaintance set, and the actor may replace its acquaintance set (as well as its script) when changing behaviour.

There is no implied ordering between the above types of action[Agh86].

An important observation to make here is that the set of possible actions for a behaviour script, when not taking further communication into consideration, is very computationally weak. It is not even Turing-complete[4]. The model achieves full computational power only through communication within communities of actors, and it has been shown that patterns of actor message passing can be used to model the traditional control structures of sequential programming languages[Hew76].

Communication

The fundamental communication mechanism of the actor model is *asynchronous buffered message-passing*, similar to the asynchronous communication primitives described in section 2.2.3. The actor communication system is sometimes metaphorically referred to as the "mail system", a metaphor coined in part to distinguish the asynchronous actor communication model from the synchronous communication models used by CSP and process calculi such as CCS and the π-calculus: In a similar spirit as the "mail system" of the actor model, one could liken the communication systems of these models to a "telephone network". The communication semantics of the actor model has the following defining characteristics[HRAA84]:

Asynchrony : Messages are not guaranteed to arrive in the order in which they were sent. This also implies that a sender does not need to pause execution to wait for an acknowledgment, but may proceed immediately.

Fairness : The actor model assumes *weak fairness*: When a message is sent, it will arrive in finite time. The model makes no other assumptions about the order in which messages arrive.

Directness : The sender of a message must have an explicit reference to the address of the message target[5].

The *only* mechanism for both control flow and data transfer in the actor model is message passing. Similar to how the LISP family of languages blurs the distinction between code and data[AS96], the actor model further blurs the distinction between control flow and data flow[Hew76]. Because of these characteristics, the actor model is able to model many different forms of data and control structures, ranging from the well-known concepts of recursion and iteration, to more esoteric forms of control and data structures, such as races[6][Lie81b].

[4]In particular, the lack of primitive recursion and iteration makes a single actor incapable of simulating a looping Turing machine without resorting to communication with other actors, or with itself.

[5]The directness characteristic is not mentioned as an explicit characteristic in actor literature, it is assumed. Directness follows from the fact that an actor may only send messages to its acquaintances, or to actors named in a message it has received.

[6]A race is a sequence ordered by the time each element took to compute.

Events and their Orderings

Fundamental to understanding concurrency and sequentiality in the actor model is the concept of *events*, and how events are ordered. In shared-memory concurrency models, the fundamental construct used to reason about the sequence of computational steps is the *global state*. In such models, computation can be understood as a sequence of transitions from one "state of the universe" to another. This view, however, is not suitable for message-passing models such as the actor model, which do not have a concept of global state. Instead, the implied sequences of actions in the actor model can be reasoned about using *event orderings*.

Events are the fundamental interactions in the actor model, each comprising the arrival of a message at a target actor. Since message passing is the only means of interaction in the actor model, events provide a useful mechanism to reason about the sequence of actions in an actor system. A set of foundational laws about event orderings are given in the seminal paper *Actors and Continuous Functionals* [HB77], the most important one of which being the observation that *causes always strictly precede effects*, just as in the physical universe. In other words, when event E_1 is the cause of E_2, then the progression of events $E_1 \rightarrow E_2$ must be strictly sequential. As in our view of concurrency presented in section 2.1, a set of events are *concurrent* when there are no causal relations between them, and they may be interleaved or parallelized arbitrarily.

Thus, there are many concurrent events in actor programs, and they are easy to recognize. For instance, if an event E is initiated by the reception of message m_{in} and, the affected actor responds by transmitting a *set* of messages M_{out}, then no message in M_{out} can be the cause of any other message in M_{out}[HB77]. The events caused by these messages are thus concurrent, and an implementation is free to parallelize or interleave them.

Order Dependence

A behaviour is said to be *order-independent* or *parallel* if its behaviour never changes. This means that no message received by an actor with such a behaviour can ever affect its response to any other message it receives. The class of order-independent actors thus includes functions, in the sense of mathematics or pure functional programming: For any application on the same input, a function always returns the same output. Because of this characteristic, when an order-independent actor receives several incoming messages, an actor implementation can process them all in parallel, instead of buffering them.

Conversely, a behaviour is *order-dependent* or *serial* if receiving one message can affect how it will respond to subsequent messages – in other words, if it has local side effects. When receiving multiple messages, an actor with an order-dependent behaviour has to process them in a strict sequence. Many objects in the object-oriented programming style fall into this category. For example, the behaviour of a bank account is strictly

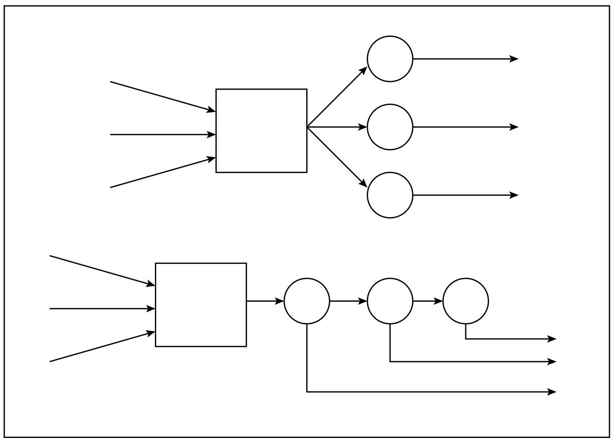


Figure 2.7: Control flow in order-independent (top) and order-dependent (bottom) actors

dependent on the order of withdrawal and deposit messages it receives. Actors that have order-dependent behaviours implicitly serialize their incoming messages, providing a means of coordination in actor programs.

Composition of Actors

The lack of Turing-completeness in a single actor has major implications for the compositionality of actor systems: It means that a composition of several actors cannot necessarily itself be an actor. For this reason, the currently accepted interpretation of the actor model[Agh85] includes the notion of *actor configurations*. An actor configuration represents a specific "community" of actors at a specific instant in time, and consists of the following components:

- A *population* of actors.
- A set of currently *pending messages*, which have not been delivered to any actor yet.
- A set of *receptionists*. The receptionists of an actor configuration is the set of all actors that are capable of receiving messages from actors *outside* of the configuration. In other words, the addresses of receptionists are known to actors outside the configuration.
- A set of *external actors*. These are actors which are not members of the configuration, but to which actors within the configuration can send messages. In other words, the addresses of external actors are known to actors within the configuration.

While actors do not compose, actor configurations do. Furthermore, since an actor configuration can effectively "hide" non-receptionist actors from the rest of the system, they can be used as a means of abstraction in actor programs, without introducing further constructs into the model. An actor within a program can abstract over a particular actor configuration through the address of one of the receptionists of that configuration.

Deadlock in Actor Systems

Because of the asynchronous message model, the classical form of "low-level" deadlock, in which a cyclic chain of blocking calls are dependent on each other finishing, does not occur in actor systems. This is true precisely because the act of sending a message does not block. However, the fact that actors are purely reactive entities implies that message reception *does* block. This means that actor systems can exhibit a behaviour similar to classical deadlocks, a phenomenon which is sometimes called *datalock* or *semantic deadlock*[Agh85]. In this thesis, we will refer to this phenomenon as *actor deadlock*.

In section 2.1.2, we saw four necessary and sufficient preconditions for deadlock to occur. However, the phrasing of these preconditions were heavily oriented towards shared-memory concurrency, and thus it can be difficult to apply them to actor systems. To give an example of using reasoning on actor semantics to understand concurrency phenomena, we therefore now develop an analogous set of preconditions for actor deadlock to occur.

Definition 3 (Actor deadlock) A configuration of actors A is in a state of *actor deadlock* when it has entered a state where all communication within the population has ceased permanently. □

Theorem 1 *Given a configuration of actors A, for A to be in a state of actor deadlock, the following conditions must hold:*

- ***Passivity**: Every actor in A is awaiting incoming messages.*
- ***Insensitivity**: Every actor in A only accepts messages from actors in A.*
- ***Silence**: A has no currently pending messages.*

Proof If every actor a in A is in a waiting state, then it follows from the reactivity property of actors that a will not initiate further communication, unless it receives a message from some actor first.

Furthermore, if every actor a in A only accepts messages from other actors in A, then it follows that a will not initiate further communication if it receives a message from an actor outside A.

However, since there are no pending messages in A at the time of observation t, then no actor a in A will receive any messages from any actor in A, unless that message is sent after t. Since all actors are in a waiting state, then no actor in A will send any messages after t. Therefore, all communication in A has ceased. ■

This suggests a general strategy for handling datalock in actor systems: Avoiding the insensitivity condition. However, as discussed in section 2.2.3, there are some situations in which insensitivity is the desired behaviour in actors. For example, if an actor is awaiting a response from another actor before it can compute its own future behaviour, it should generally not accept other messages than the response. However, depending on the situation, it may be safe to respond to *specific* messages from the outside world. For example, the actors could enter "mostly insensitive" states, in which they still accept messages from some external actor – for example, a periodic timer. When such a message arrives, the actors can take appropriate action to recover from the datalock. In the dining philosophers example, this would mean that after a philosopher has waited for some time, holding one chopstick, he could politely put it down, or try to talk another philosopher into dropping his.

2.3.2 History of the Actor Model

The actor model originated in the MIT Artificial Intelligence research community in the early 1970s. Unlike most other computational models from that era, it was explicitly designed to allow for concurrency and parallelism, because of the philosophical view that complex systems are best modelled as societies of cooperating simple individuals[GH75], rather than as a single, complex individual. A distinctive trait of the early research in the model was that actor research methodology drew inspiration from the laws of physics, as much as from mathematical logic: Early actor model researchers made observations on the nature of parallel and concurrent computing systems, and formulated universal laws based on these observations[HB77]. Throughout its history, the model has seen significant coevolution with both process calculi and early object-oriented programming languages.

The first usage of the term "actor" in the literature is in Carl Hewitt's 1971 Ph.D thesis[Hew71], detailing the development of the Planner programming language. Here, the term denoted active program elements which were triggered by the presence of specific patterns in a knowledge base. In *Actor Induction and Meta-Evaluation*, published in 1973, the term had evolved to mean something more akin to the term "agents" in distributed artificial intelligence.

The first major theoretical milestone in the formal development of actor theory came with Irene Greif's 1975 Ph.D thesis[Gre75], which introduced the event diagram formalism for actors, and gave a formal operational semantics for the model. In one of the seminal papers on the actor model, the 1976 article *Viewing Control Structures as Patterns of Passing Messages*[Hew76], Hewitt demonstrated that by applying an actor-based variant of the continuation-passing style of programming, all the familiar sequential programming control structures could be encoded as message communication patterns in actor systems. As part of his work with the model, Hewitt developed the programming language Plasma [HS75], a Lisp-based interpreter meant for practical experimentation with the semantics of actor programs.

Following up on Greif's thesis work, Hewitt and his student Henry Baker formalized a set of universal laws for concurrent computation in *Actors and Continuous Functionals*[HB77], published in 1977. Many of these laws deal with the orderings and causal relations of events, formalizing the conditions which guarantee that causes must always strictly precede effects.

Another major milestone was the introduction of *guardians* and *serializers*[AH78, HA79], to address the need for a means of coordination when modelling order-dependent actors. Both of these notions were inspired by C.A.R. Hoare's work on *monitors* [Hoa74], a construct for selective suspension and resumption of processes in shared-memory concurrency models. In 1981, Will Clinger developed a denotational semantics for the actor model, based on power domains[Cli81]. This work showed the consistency of the axiomatic laws previously identified by Hewitt and Baker. Throughout most of the early 1980's, the primary implementations of the actor model was the Act family of languages[Lie81a, The83, HA86], which included many novel (but also quite complicated) language features.

The currently accepted formulation of the actor model was developed in Gul Agha's 1987 Ph.D thesis[Agh85]. In this thesis, Agha developed a transition-based semantics for actors, and introduced composability into actor computation through the development of the concepts of configurations, receptionists and external actors. In 1987, Carl Manning developed Acore [Man88], which was the first actor implementation based on Agha's formulation of the model. The model itself has remained reasonably stable since then, although some work has been done in integrating it with sequential computation models. For example, in *Foundations of Actor Semantics*[Cli81], Agha et.al developed a theoretical actor language based on a combination of the actor model with the λ-calculus.

Relatively little actor-related work was done in the 1990s, although the advent of multicore processing as well as massively distributed systems sparked some renewed interest in the model. In the late 1980s and throughout the 1990s, a group at Ericsson Computer Science Laboratory, led by Ulf Wiger and Joe Armstrong, developed the Erlang language for robust telecommunication applications[Arm03]. While Erlang is not explicitly based on the actor model, its notion of processes is very similar to actors. Other actor implementations of the 1990s and early 21st century include the E language[ELa], the Scala actor library[HO07], as well as concurrency model of the Io scripting language[Dek05].

2.4 Summary

Although concurrency is useful in many problem domains, concurrent programming is fundamentally more difficult than sequential programming. This additional difficulty stems from the fact that a concurrent program can have several possible interleavings, and that a concurrent program is correct only if it is correct for *all* possible interleavings.

Concurrent computation models help reduce this additional difficulty by placing constraints on the ordering of computational steps in concurrent program executions, thus reducing the number of possible interleavings. The universe of concurrent computation models is very diverse, and covers both deterministic, nondeterministic, shared- and distributed-memory models, as well as many different mechanisms of expressing synchronization. We saw that a nondeterministic, message-passing, asynchronous model fit well with our goal of designing a simple, general-purpose, high-level concurrent language.

We investigated one example of such a model: The actor model. The actor model is conceptually a quite simple model, and can serve as the theoretical basis of a high-level concurrent programming language. Because of the few and relatively simple concepts in the model, it appears to be a good way for programmers to conceptualize concurrent programs, in no small part because it promotes some of the classical concurrency problems to the program design level, rather than leaving them as implementation details.

Having investigated some of the underlying theoretical issues of concurrent programming, we now turn our attention to the practical issues in the development of such a language.

Recommended Literature

One of the first fields within computer science to consider the problems of concurrency was that of operating systems research, because this field has historically always had to consider the situation of several processes sharing a single machine. Many operating systems textbooks give good treatments to concurrency, John English's *Introduction to Operating Systems: Behind the Desktop*[Eng04] is a good example. Two seminal papers on concurrency in this context are Hoare's *Monitors: An Operating System Structuring Concept*[Hoa74], as well as Dijkstra's *Hierarchical ordering of sequential processes*[Dij71].

In *Concepts, Techniques and Models of Computer Programming*[RH04], Peter Van Roy and Seif Haridi give an excellent introduction to concurrency models, using the multiparadigm Oz language to illustrate the concepts. This book demonstrates both deterministic, message-passing and shared-memory models of concurrency. It also demonstrates several sequential computation models, and gives good examples of their use.

Gul Agha's Ph.D thesis *Actors: A Model of Concurrent Computation in Distributed Systems*[Agh85] gives a very solid treatment of the foundations of the actor model – and is a quite enjoyable, accessible read as well. The two seminal actor papers *Viewing Control Structures as Patterns of Passing Messages*[Hew76] and *Actors and Continuous Functionals*[HB77], while somewhat dated, are also particularly enlightening reads, and invite the reader to consider a new angle on the nature of some of the programming constructs most practicing programmers have come to take for granted. For a somewhat lighter introduction to actor programming, Agha's *An Overview of Actor Languages*[Agh86] is highly recommended.

3. The Merlin Language

Human beings do not live in the objective world alone, nor alone in the world of social activity as ordinarily understood, but are very much at the mercy of the particular language which has become the medium of expression for their society.
– Edward Sapir (1884–1939)

Language is the source of all misunderstandings.
– Antoine de Saint-Exupery (1900–1944)

In his 1972 Turing Award lecture, *The Humble Programmer*[Dij72], Edsger Dijkstra gave an eloquent argument that it was of vital importance for the programming community to pay closer attention to intellectual manageability in programs. He criticized the excessive complexity of programming languages, not for technical or mathematical reasons, but for *psychological* reasons: The human mind has a limited capacity to manage complexity, and competent programmers are aware of their own limitations. He argued in favour of "modest and elegant programming languages", and near the end of his lecture, he made an observation that was considered highly controversial at the time:

> I observe a cultural tradition, which in all probability has its roots in the Renaissance, to ignore this influence, to regard the human mind as the supreme and autonomous master of its artefacts. But if I start to analyse the thinking habits of myself and of my fellow human beings, I come, whether I like it or not, to a completely different conclusion; that the tools we are trying to use and the language or notation we are using to express or record our thoughts, are the major factors determining what we can think or express at all.

Programming languages are tools for creating programs. But they are tools which profoundly shape even our *thoughts* about programs, and the successful design of such a tool is not, by any means, a simple task. Because of the influence of programming languages on the thinking habits of the programmers who use them, the programming language designer must himself be acutely aware of the limited human ability to manage complexity. This especially holds true for designers of concurrent languages – as we have seen in section 2.1, concurrent programming is *inherently* more complex than sequential programming. The central challenge underlying the design of concurrent high-level languages is the development of linguistic abstractions for intellectually managing programs which are subject to nondeterminism.

Chapter Structure

This chapter consists of three sections:

- *From Model to Language*: The first section of the chapter gives an overview of the philosophy and background for the MERLIN language. It introduces the overall design goals of the language, chief of which is simplicity. The section gives a brief exploration of the meaning and value of simplicity in programming languages, and concludes with a set of design principles, which can be used to select proposed language features later in the process.

- *Introduction to* MERLIN : This section gives an introduction to the MERLIN programming language, with a focus on demonstrating all the linguistic concepts used in the language through illustrative examples. Linguistic concepts are explained and rationalized as they are introduced.

- *The Design of* MERLIN : This section of this chapter discusses the design process of the MERLIN language. It describes some of the challenges in developing an actor-based language, and discusses the most important design decisions made during the development process. This chapter takes a look at some of the roads not taken.

- *Evaluation of the* MERLIN *Language Design*: This section considers the MERLIN language design, contrasted with the design goals. It also discusses the relationships between MERLIN and other actor-based programming systems.

3.1 From Model to Language

As we have seen, one of the appealing qualities of the actor model is its conceptual simplicity. There is one fundamental type of concurrent computing entity (the actor), and one universal means of communication (the asynchronous message). However, the actor model is a computational model, not a programming language. As such, it leaves many questions unanswered. For example, it lacks a name binding mechanism, a means of abstraction, a means of combination, and a representation for behaviour scripts. Thus, even after making the decision to base a concurrent language on the actor model, the language designer faces many open questions.

Because programming languages are tools of thought, the language design process must pay some attention to the nature and process of thought. This necessarily involves considering some subjects which are sometimes considered outside the domain of computer science, and within the domain of philosophy. However, in programming language design, there is a strong link between these two domains precisely because the language designer must be aware of the thinking habits the language promotes.

3.1.1 Design Philosophy

The MERLIN language represents an attempt to develop a simple concurrent programming language, based on the actor model. As with all languages based on a formal computational model, the design of MERLIN involves a process of balancing two concerns which are often in direct conflict: *Conceptual purity* and *practical applicability.* For all the desirable qualities we have identified in the actor model, it is a computational model, not a programming language. A straightforward implementation of the model would be a very low-level language, lacking many facilities a modern programmer expects from an expressive, high-level programming language. Therefore, MERLIN is more correctly described as a concurrent language *based on* the actor model, as opposed to an *implementation* of the actor model. One of the most important tasks in the design process of MERLIN is to effectively bridge the gap between abstract model and concrete programming language.

Simplicity: Economy of Thought

Simplicity is frequently mentioned as a particularly desirable quality in programming languages[Seb03, Dij72], although relatively little attention has traditionally been paid to what it *means* for a programming language to be simple. Since we have observed that concurrent programming has a higher inherent complexity than sequential programming, it seems *especially* important for a concurrent language to avoid introducing any unnecessary complexity.

A useful guiding principle regarding simplicity is the philosophical principle of *Occam's Razor*, a principle originally developed in the 14th century by the logician and Franciscan monk William of Ockham. This principle simply states:

> It is vain to do with more what can be done with less.

In most contemporary interpretations of Occam's Razor, it is taken to mean that when all else is equal, it is rational to prefer a formulation involving few entities than one involving many. Indeed, many modern paraphrases of the principle use the alternate wording "entities should not be multiplied beyond necessity". Because of the wide acceptance of the Razor within philosophy and scientific methodology, it seems reasonable to adopt its underlying idea as a basis for judging simplicity in a programming language. But *which* entities should we avoid multiplying beyond necessity?

According to Dijkstra[Dij72], the scarcest resource in programming projects has long been the brainpower of programmers – not because programmers are unintelligent, but because they are often working on very difficult problems. This perspective leads to a view of simplicity as a principle of *economy of thought*: Thought is a valuable resource, and should be used as efficiently as possible. Furthermore, because programmers often work on difficult problems, it stands to reason that they should be able to devote as much thought as possible to solving those problems, rather than dealing with the intricacies of their programming tools. Therefore, we seek a language with a *light mental footprint*.

The economy of thought principle can be applied to the programming language design itself, yielding a derived principle of *economy of concepts*. Since each individual fundamental concept within a language adds to the overall mental footprint of a language, linguistic concepts themselves should be regarded as a scarce resource, and thus should be used sparingly. However, it is not enough to have *few* of them: They must have *high expressive power* as well[1].

Consistency: The Principle of Least Surprise

An important method of reducing the mental footprint of a programming language is by paying very close attention to the overall *consistency* of the language design. Consistency lightens the mental footprint of a programming language by allowing a programmer to remember a set of fundamental rules, which the rest of the language follows. Thus, *inconsistencies* are linguistic elements, whether syntactic or semantic, which violate such fundamental rules. As an example of a syntactic inconsistency, the standard library of the PHP language includes several search facilities, but the positions of the needle and haystack arguments in these functions follow no rule, forcing the programmer to memorize the positions for each individual function – or be prepared to consult the documentation frequently. Consistency in programming languages can also encourage a *uniqueness* principle[Mey00]: A consistent language should provide one, and only one, good way to perform any action of interest. Adhering to this principle further lightens the mental footprint.

Abstraction: Scaling Simplicity

As programs grow in size, they inevitably and naturally grow in complexity. Thus, the economy of thought principle must be made scalable. Most programming languages do this by providing linguistic support for the fundamental mechanism the human mind uses to cope with increasing complexity: Abstraction. The use of abstraction allows programmers to design his program as a hierarchy of conceptual layers, such that elements in each layer is defined in terms of the elements of the layers below it. The lower

[1]An extreme example of a language with few concepts is the Brainf*ck language, developed in 1994 by Urban Müller. Brainf*ck has only one datatype (the byte) and only eight commands – two for incrementing and decrementing, two for I/O, two for pointer arithmetic and two for loops. Even with this extremely small set of concepts, Brainf*ck has been proven to be Turing-complete. Whether or not Brainf*ck can be said to have a light mental footprint is left for the reader to decide, while pondering the Brainf*ck version of the classic Hello World example:

```
++++++++++
[>+++++++>++++++++++>+++>+<<<<-]
>++.>+.+++++++..+++.
>++.<<+++++++++++++++.>.+++.---.----.>+.>.
```

levels in such an architecture have more detail, whereas the upper levels have higher expressiveness: They allow more to be done with less entities, since the programmer can ignore the internal details of elements in lower levels. Essentially, abstraction mechanisms introduce Occam's razor into the programmer's tool set. Good abstraction mechanisms allow programs to maintain a light mental footprint even as they increase in size.

Thus, the fundamental ideal MERLIN strives for is, above all, maintaining a light mental footprint. This, and the three means (simplicity, consistency and abstraction) proposed for achieving this goal, makes up a statement of philosophy for the MERLIN language: They give a high-level, conceptual vision for the language design. We now turn our attention to developing a more concrete formulation of language requirements based on this vision.

3.1.2 Language Requirements

Above all, a programming language is a tool for reading, writing, and thinking about programs. Therefore, the requirements of a programming language can be viewed in terms of the qualities it should encourage in the programs expressed in it. The list given here is inspired by an almost similar list in Robert W. Sebesta's textbook *Concepts of Programming Languages*[Seb03]. The *efficiency* quality has been added to Sebesta's list.

Readability : The *readability* of a program determines how easily it can be read and understood by someone who is reasonably well-versed in the language it was written in. Language qualities that foster readability include *simplicity*, *expressivity* and *consistency.*

Writability : The *writability* of programs is a measure of how quickly and "painlessly" a programmer who is reasonably well-versed in the language can write programs in it. Language features that improve writability include all readability characteristics, since writing programs involves reading the program as it is written. An additional characteristic that specifically improves writability is *composability.*

Reliability : The *reliability* of a program indicates which guarantees it can provide that it will perform to specification, without execution errors. Language features that improve reliability include all the readability and writability qualities, since clear programs conceal less errors than incomprehensible ones, and programs that are difficult to write will leave more room open for programmer error. Furthermore, reliability is improved by facilities for *static analysis.*

Efficiency : The *efficiency* of a program determines its ability to minimize its use of system and hardware resources. Language qualities that improve efficiency include all the readability and writability qualities, since it is easier to implement high-performance algorithms in a language that allows such algorithms to be written clearly. The reliability-related capacity for static analysis can also improve efficiency, since static analysis can be used by compilers to perform code optimizations. Finally, *low overhead* language elements improve program efficiency.

A list of language requirements can be assembled from the characteristics that foster the desired qualities in programs written in the language. Because of the ideal of keeping a light mental footprint, the Merlin project focuses on readability and writability. A convincing case can be made that reliability lightens the mental footprint as well, since static analysis facilities simplifies the debugging process. However, the integration of static analysis facilities into a language implies a static type system to place constraints on the dynamic behaviour of programs. The actor model strongly emphasizes dynamicity and flexibility, which means that by introducing a static type system, we risk introducing inconsistencies with the model. Whether static or dynamic type systems have the lightest mental footprint is a frequently debated topic in the programming language design community, which appears to have no simple answer.

The desired qualities of the language can be gathered in an ordered list, to assist in evaluating proposed linguistic concepts. However, programming language design is fundamentally a balancing act: All the requirements for programming languages represent desirable qualities, but they are also often in direct conflict with each other[Seb03]. Furthermore, because any design process involves a certain degree of aesthetics, requirements for programming languages are by nature a controversial topic. The following design requirements for Merlin primarily follow from the language philosophy, but also necessarily reflect the author's subjective sense of programming aesthetics.

1. **Simplicity**: A language construct is *simple* if its description and comprehension involves few concepts.

2. **Consistency**: A language construct is *consistent* if it does not cause any unpredictable interactions with other language constructs.

3. **Expressivity**: A language construct is *expressive* if it allows expressing complex concepts with a minimum of syntactic entities.

4. **Composability**: A language construct is *composable* if it can be freely combined with other language constructs, to form complex structures.

5. **Static Analysis**: A language construct supports *static analysis* if it is possible to guarantee, prior to executing the program, that the construct will not be used in error.

6. **Low Overhead**: A language construct has *low overhead* if it can be mapped to underlying system primitives with only a minor increase in complexity.

Simplicity, *consistency* and *expressivity* are given the highest priorities, because these qualities follow directly from the design philosophy. *Composability* is given medium priority because a language based on the composition of simple, consistent units, combined with a good abstraction mechanism, scales well: The same basic linguistic means that can be used to express simple structures can also be used to describe complex systems composed of those structures[AS96]. *Static analysis* is given low priority, for the reasons given above. Finally, *low overhead* is given the lowest priority: Merlin is allowed to waste computer time, in order to save programmer time.

This concludes the conceptual design considerations for Merlin. The following section gives an introduction to the Merlin language, while section 3.3 describes issues and decisions that arose during the design process.

3.2 Introduction to Merlin

Merlin is a concurrent, actor-based programming language. It is in many respects an unusual language, mostly due to its strong emphasis on concurrency. The purpose of this section is to introduce the syntax and semantics of Merlin programs, by showing and describing a collection of Merlin programs. The small programs in this section were not chosen to be efficient or even particularly useful, they are intended merely to show the essence of programming in Merlin. The syntax of Merlin, given as an EBNF grammar, can be found in Appendix A, and an extensive discussion of the rationales behind the language design choices is given in section 3.3.

3.2.1 Inspirations

No programming language was developed in a vacuum. Apart from getting most of its theoretical foundation from the actor model, two languages that had a significant influence on the design of Merlin are Erlang and Scheme. Like Scheme, Merlin emphasizes the use of dynamic, first-class entities, and uses only a single namespace. Like Erlang, Merlin is heavily oriented towards message-passing concurrency, and uses a message reception scheme based on pattern matching.

3.2.2 Introductory Examples

Honouring a very long computer science tradition, our first Merlin program has a simple and not too surprising purpose:

```
tell console "Hello World!"
```

This small, traditional example nevertheless shows an important concept of Merlin: Actor communication. The entity `console` in the example is the address of a *built-in* *actor*, which will display any message sent to it on the console. Messages are sent using the `tell` statement, the first argument of which must be a valid actor address, and any subsequent arguments are sent to that actor as a message. Messages are a very important concept in MERLIN, because actors in MERLIN *only* carry out any activity in response to receiving a message. We refer to the interval of time between the reception of a message, and the completion of its processing by the receiving actor, as an *event*.

Actors are defined by their *behaviours*, which are said to be *instantiated* when an actor is created from that behaviour. We can demonstrate how to define behaviours and create actors by implementing a very simple actor, which functions as a proxy for `console`:

```
con-proxy : [[msg | tell console msg]]
cproxy : spawn con-proxy ()
tell cproxy "Hello World!"
tell cproxy 42
```

The `con-proxy` behaviour is a very simple one. Its script, `[[msg | tell console msg]]`, should be read: *For any message `msg`, tell the console `msg`.* The expression `spawn con-proxy ()` creates a new actor from the behaviour given in `con-proxy`.

The `:` operator is used to bind names to values. This has the interesting implication that behaviour specifications and actor addresses are *values*, just like numbers and strings. Therefore, if the `con-proxy` behaviour will not be needed later in the program, the first two lines could have been written thus:

```
cproxy : spawn [[msg | tell console msg]] ()
```

Similarly, if the `cproxy` actor was only needed once, then the example could have been written in a single statement:

```
tell spawn [[msg | tell console msg]] () "Goodbye Cruel World!"
```

This style can lead to more concise programs, but is not always particularly readable.

Data Types in Merlin

In the console proxy example, we spawned the actor `cproxy` and sent it two messages, the string `Hello World!` and the number 42. Both were temporarily bound to the name `msg` when they were received by `cproxy`, before being sent off to `console`. This shows another important characteristic of MERLIN programs: They are *dynamically typed.* This means that type information is not associated with names, but with values, and that the types of MERLIN values can only be known at run time. We can make a variant of the proxy which only forwards strings:

Data Type	Type Predicate	Operations
Number	`number?`	`is + - * / = > < >= <=`
Boolean	`boolean?`	`is and or not`
String	`string?`	`is {a:b} .`
Tuple	`tuple?`	`is {a:b} .`
Address	`address?`	`is`
Behaviour	`behaviour?`	`is spawn`

Table 3.1: Data Types and Operations in the Merlin language

```
str-proxy : [[msg, string? msg | tell console msg]
             [msg | tell console "Type error in str-proxy!"]]
sproxy : spawn str-proxy ()
tell sproxy "Hello World!"
tell sproxy 42
```

The `sproxy` actor behaves just like the `cproxy` of the previous example when sent the string "Hello World!". When sent the number 42, however, it instead causes an error message to be displayed.

The expression `string? msg` is an example of a run-time type check in Merlin. The unary operator `string?` is a *type predicate*, which returns true only in the case where its argument belongs to the specific type it checks for. Each of the Merlin built-in types have a type predicate, as well as a set of operators defined on them. Table 3.2.2 gives the full set of Merlin data types, their type predicates, and their operators.

3.2.3 Pattern Matching

An important detail in the type-checking console proxy is that unlike the `con-proxy` behaviour, the `str-proxy` behaviour is defined so that it responds to two different kinds of messages. Each of the two blocks `[msg, string? msg | ...]` and `[msg | ...]` is called a *receptor*. The left-hand side of the vertical bar `|` is called the *pattern*, and the right-hand side is the *reaction*.

Behaviours may define as many receptors as they require. When an actor receives a message, the message is tested against the patterns of the receptors sequentially, and the first receptor to generate a match will be selected for execution. The following program gives a somewhat more detailed example of pattern matching.

```
v : 42
patterns : [[(v, n)             | tell console "Match 1: " . n]
            [(666, n)           | tell console "Match 2: " . n]
            [(n, n, 10)         | tell console "Match 3: " . n]
            [("curry", (v, v)   | tell console "Match 4."]
            [n                  | tell console "Match 5: " . n]]
pats : spawn patterns ()
tell pats (42, 10)          -- displays "Match 1: 10"
tell pats (666, v)          -- displays "Match 2: 42"
tell pats (5, 5, 10)        -- displays "Match 3: 5"
tell pats ("curry",(42,42)) -- displays "Match 4."
tell pats ("curry",42,42)   -- displays "Match 5: ("curry", 42, 42))"
tell pats 200               -- displays "Match 5: 200"
```

As this example shows, patterns consist of tuples of names and literals, and may also contain sub-tuples (the tuple delimited by parentheses in the fourth pattern, `(v, v)`, was an example of a tuple). Notice that names in patterns may be *bound* (as `v` is in the example), in which case their values are substituted into the pattern, or *free*, in which case they will become bound within the code of the reaction if a pattern matches.

Patterns may optionally include a *guard expression*, which is delimited by a comma. We have already seen an example of a guard expression in the type-checking console proxy, in its receptor `[msg, string? msg | ...]`. If a guard expression in an otherwise matching pattern evaluates to false, then the pattern is considered non-matching and the reaction is *not* executed.

3.2.4 Concurrency

Actors in MERLIN are fully concurrent and autonomous. Consider the following program:

```
ping : [["go" | tell console "ping"; tell self "go"]]
pong : [["go" | tell console "pong"; tell self "go"]]

ping-actor : spawn ping ()
pong-actor : spawn pong ()

tell ping-actor "go"
tell pong-actor "go"
```

The `self` reserved word is simply a mechanism for actors to refer to their own address. So, as expected, the two actors will send a furiously fast, infinite stream of `ping`s and `pong`s to the console. However, because of the inherently nondeterministic nature of timing in concurrent programs, the console output is *not* an ordered stream of alternating pings and pongs, but the chaotic jumble we would expect from two completely independent, uncoordinated entities simultaneously trying to communicate with the console.

Asynchrony

Consider the previous program, but with the two first lines changed to this:

```
ping : [["go" | tell self "go"; tell console "ping"]]
pong : [["go" | tell self "go"; tell console "pong"]]
```

With this behaviour, the actors tell themselves to go immediately after receiving their initial messages, and send their messages to the console afterwards. It seems reasonable to ask if the actors will ever get around to sending anything to the console.

The answer, in fact, is yes: The modified version will exhibit a behaviour observably similar to that of the original. This is because of an important characteristic of the Merlin tell statement: It is asynchronous[2]. A tell does not wait for an acknowledgment, it merely sends a message and continues.

Because of the asynchrony, Merlin actors *buffer* incoming messages. In fact, the concrete action taken by tell can be viewed as placing a message in the buffer of the target actor. Thus, what goes on in both versions of the ping-pong example is that the actors place a new go message in their own buffers, place a ping/pong message in the buffer of console, and continue.

We can also make another observation: Actors in Merlin receive messages implicitly. When a reaction has finished, the actor enters a waiting state, awaiting the next message. In the case of the ping-pong example, it will already have placed a message there itself, which it immediately begins processing.

Coordination

We observed that the two actors in the ping-pong example caused an unordered stream of pings and pongs to be sent to the console, and that this lack of order was because the two actors are concurrent, independent entities. However, what if we *wanted* them to display their output in an alternating order? We could write a version in which the two actors coordinate their activities by passing messages between each other. In this version, rather than each actor causing itself to send another message to the console, the two actors will cause *each other* to do so. We thus create a causal relation between them, such that a sequence will occur between their actions.

[2]In fact, had tell been synchronous, then the previous version of the ping-pong program would only send one ping and one pong to the console. Then it would enter a deadlock, as discussed in section 2.2.3.

This program implements that idea:

```
ping: [[("go", other) |
    tell console "ping"; tell other ("go", self)]]
pong: [[("go", other) |
    tell console "pong"; tell other ("go", self)]]

ping-actor: spawn ping ()
pong-actor: spawn pong ()

tell ping-actor ("go", pong-actor)
```

These two actors now *synchronize* their activities, with the exchange of ‘‘`go`’’ messages serving as acknowledgments. However, while this program gives the desired output in the Merlin implementation described in chapter 4, this is in fact *incidental* to an implementation detail in that interpreter, and should not be assumed. In actor semantics, we cannot assume that messages arrive in the same order as they are sent, so while the desired, alternating sequence exists at the *sending* side of the messages, it is not guaranteed that the *receiving* side will observe this sequence too. We revisit this problem in section 3.2.6.

3.2.5 Parameters

Many readers will probably have observed that the definitions of the actors in the preceding ping-pong examples were completely identical, except only for the word they passed to the console. More complex programs would quickly grow to unmanageable heaps of text if it was not possible to abstract over common trends in program elements. For Merlin actors, this form of abstraction can be achieved using parameters:

```
ping-pong : [(word)
              [("go", other) | tell console word
                                tell other ("go", self)]]

ping: spawn ping-pong ("ping")
pong: spawn ping-pong ("pong")

tell ping ("go", pong)
```

The parameter mechanism allows actors to store *persistent* values. Parameters exist across events, and in the above program, the two actors’ `word` names will remain set to `"ping"` and `"pong"` respectively, for the entire lifetime of the actors.

3.2.6 Changing Behaviour

All the actors we have seen so far have been unchanging in time, forever doing the same actions when given the same input. In this respect, these actors have been similar to mathematical functions: They map domains of messages to ranges of reactions. Many useful programs can be written using only actors with such a behaviour, and it is generally good practice to use such actors when possible: They are easy to reason about, precisely because their outputs are predictable given their inputs.

However, some problems are more fluently solved using actors that can change state over time. For example, consider an actor which counts the number of messages it has received:

```
counter : [(count)
            [ msg | tell console "Messages Received: " . count + 1;
                    become counter (count + 1)]]

count-actor : spawn counter (0)
tell count "Hello!"     -- displays "Messages Received: 1"
tell count 42           -- displays "Messages Received: 2"
```

This actor introduces the `become` statement, which does exactly what we needed: It provides actors with a mechanism for modelling state changes over time. In this example, the counter essentially becomes a *new* counter, bound to the same address, with its `count` parameter set to a new value. The `become` mechanism allows actors to instantiate new behaviours for themselves to use for subsequent messages.

The `become` mechanism is quite powerful, since it allows an actor to change its parameters *and* its behaviour scripts. If the counter in the example executed `become con-proxy` instead, all subsequent messages would be processed using the `con-proxy` behaviour. While this particular construction might make little sense, the ability to change behaviour more radically than merely changing parameters can be very useful in practice. For example, an actor which "dies" can change its behaviour to one that silently drops all messages. A `working-computer-scientist` can become a `sleeping-computer-scientist` if he receives too many `write-paper` messages, and not enough `drink-coffee` messages.

The Message Buffer, Revisited

Now that we have introduced `become`, we can take a more in-depth look at the message buffer mentioned in section 3.2.4, and how it can be used as a receiver-side synchronization mechanism. Consider the following program:

```
seq : [[(1, msg) | tell console "First step"
                   become
                     [[(2, msg) | tell console "Second step"]] ()]]

seq-actor : spawn seq ()

tell seq-actor (2, "hello")
tell seq-actor (1, "hello again")
```

This program will cause the messages `First step` and `Second step` to appear on the console – *in that order*. This may at first seem strange, because of the order it was sent its messages in. However, when the actor receives the message `(2, "hello")`, it has no matching receptor for the message. It places the message in its buffer. When it subsequently receives the message `(1, "hello again")`, it changes its behaviour to one which *does* have a receptor for the buffered message. At this point, all buffered messages are checked if they match any receptors in the new behaviour, and processed if they do.

This can be useful for maintaining an order of actions in the face of incoming messages arriving in an unpredictable order. Since we have seen that MERLIN actors are independent, concurrent entities with asynchronous communication facilities, unpredictable message arrival can be a reasonably common situation (the uncoordinated ping-pong program gave an example). The combination of `become` and the message buffer provides a means to cope with it.

3.2.7 Recursion

We have already seen an example of recursive communication patterns, in the first ping-pong example. This was a quite simple example of recursion, in which the actors merely continually told themselves to continue. We can also model other recursive programs:

```
fac : [[(1,t) | tell t 1]
       [(n,t) | tell self (n - 1, spawn [[m | tell t (m * n)]])]]

factorial : spawn fac ()

tell factorial (10, console)
tell factorial (20, console)
```

This small program computes 10! and 20!, and sends both to the console. Its somewhat strange appearance is due to the lack of functions and call stacks in MERLIN. Because of this important omission, MERLIN programs must explicitly maintain the data across the levels of the recursion. The program above is written in a form somewhat similar to *continuation-passing style*, which is sometimes used as an intermediate representation for compilers in functional languages. The `factorial` actor simulates a function call stack by spawning a chain of new actors, each of which will carry out the "rest of the computation" after the tell.

Another characteristic of MERLIN programs demonstrated in the factorial actor is *static scope*: When the continuation actor behaviour is defined by the `factorial` actor, it refers to the names `t` and `n`. The values of these names are bound to their values in the *lexically enclosing block* – in this case, `factorial` itself.

Event Atomicity

An important observation to make at this point is that events in MERLIN are *atomic*. Once an actor accepts a message, it is *guaranteed* that at some point in the future, it will finish processing it. This guarantee stems from the fact that *all* actions of MERLIN behaviour scripts block for only a bounded span of time, and the fact that any single reaction may carry out only a finite number of actions. This guarantee is possible because there are no recursion or iteration constructs built into MERLIN, and because of the asynchronous communication. This means that without the communication system, MERLIN is *not* Turing-complete, since iteration and recursion is *only* possible using messages.

3.2.8 Merlin Overview

We have seen a selection of examples, demonstrating the important concepts of the MERLIN language. We can sum up what we have seen in a set of eight core precepts, describing the defining points of MERLIN:

- **Actor-Based**: Programs are structured as communities of autonomous actors.
- **Message Passing**: Actors can only interact by message passing.
- **Asynchrony**: Message sending does not block.
- **Concurrency**: Multiple actors may carry out activities simultaneously.
- **Reactivity**: All activity is triggered by events, at message reception.
- **Event Atomicity**: All events are guaranteed to eventually terminate.
- **Pattern Matching**: Message responses are selected by pattern.
- **Buffering**: Received messages are buffered until recognized.

These precepts concisely sum up the MERLIN "view of the world". They were a central principle in the design process of MERLIN, and are discussed in greater detail in the following section.

3.3 The Design of Merlin

The Merlin language represents an attempt to realize the goal of developing a concurrent language with a light mental footprint. Several issues and challenges came up during the development of Merlin, and this section discusses how and why these were met.

3.3.1 The Core Precepts

One of the first design decisions in the Merlin design process was to develop a concise summary of the "view of the world" the language should represent. The primary reason for this step is *maintaining consistency*: By having a set of concrete precepts to represent the vision behind the language, each linguistic concept can be contrasted with the precepts. Taking this view, an inconsistency is simply a linguistic concept that violates one or more of the precepts. Let us investigate each of the precepts.

- **Actor-Based**: Programs are structured as communities of autonomous actors.
  Rationale: This precept largely follows from the choice of basing the language on the actor model. There *are* elements in Merlin programs which are not actors (see section 3.3.2), but these are all *passive* entities which cannot carry out computation or send messages.

- **Message Passing**: Actors can only interact by message passing.
  Rationale: This precept implies that there can be no observable shared memory. We discussed the strengths and weaknesses of message-passing and shared-memory concurrency in section 2.2. *Mixing* the two models could be quite confusing, and would violate the uniqueness principle discussed in 3.1.1.

- **Asynchrony**: Message sending does not block.
  Rationale: We discussed the respective benefits and drawbacks of asynchronous and synchronous communication in section 2.2. The primary benefits of non-blocking, asynchronous message sending are increased concurrency, and that recursive communication patterns are possible.

- **Concurrency**: Multiple actors may carry out activities simultaneously.
  Rationale: This follows from the nature of Merlin as a concurrent language.

- **Reactivity**: All activity is triggered by events, at message reception.
  Rationale: It is possible to make an *active* variant of actors, which may also perform activities apart from message reactions. However, these would be significantly more difficult to reason about. The definition of actor deadlock in 2.3 would not hold for such entities, and it would also be more difficult to determine whether or not such an actor has become garbage.

- **Event Atomicity**: All events are guaranteed to eventually terminate.
  RATIONALE: Combined with the reactivity precept, this makes MERLIN actors *considerably* more easy to reason about. If an actor could take infinite time processing a single event, then other actors awaiting messages from that actor could also be held up for infinite time.

- **Pattern Matching**: Message responses are selected by pattern.
  RATIONALE: Combined with message queues for unrecognized messages, pattern matching provides a powerful means of enforcing an order in received messages. Such an order cannot necessarily be assumed in a system with concurrent, independent entities, and the semantics of the actor model specifically state that such an order cannot be assumed.

- **Buffering**: Received messages are buffered until recognized.
  RATIONALE: See above. Furthermore, some form of buffering is required in asynchronous communication, or messages received while an actor is busy would be lost.

As already mentioned, the purpose of the core precepts is to ensure against inconsistencies: Any proposed language element which violates one of the precepts should be considered an inconsistency, and not allowed in the language.

3.3.2 Pure Actor Languages

MERLIN is an *impure* actor language, in the sense that it supports primitive data types which are not actors. There are several reasons for this choice, some of them of a philosophical nature, others more pragmatic.

It is clearly *possible* to design a pure actor language, in which *everything*, right down to the humblest integer, is modelled as message-passing actors. In such a model, even *messages* can be modelled as actors – in fact, in older actor literature[HB77], messages are referred to as *messengers*, for precisely this reason. This variant of the actor model is sometimes referred to as the *universe of actors* model[Agh85]. Several pure actor languages have historically been implemented, notably the ACT family of languages[Lie81a, The83, HA86], which long served as the "reference" implementations of the actor model. Since pure actor languages represent many phenomena with just a single basic type of entity, should they not be considered conceptually simpler than impure languages? Let us consider the exact implications of actor purity.

Infinite Regress

A fundamental problem pure actor languages have to deal with is that of *infinite regress*: If everything is an actor, and actors are limited to the four simple actions given in the formal definition of actors (conditional expressions, spawning new actors, sending messages and changing behaviour – see section 2.3), then how can a program ever perform any actual computation?

In particular, consider the situation if messages are actors. The only defined means of interaction in the actor model is message passing, so when a message actor arrives at its target, the only way it can interact with the target is by sending messages to it. And when these messages arrive, the only way *they* can interact with the target is by sending messages to it. And when ...

This problem can be solved by introducing a notion of *primitive actors*[Hew76], sometimes referred to as *rock-bottom actors*[Lie81a]. These actors have the special ability to perform work without sending further messages into the system. In particular, they can obtain the acquaintances of any messenger actors they receive *without* any message interaction. This solves the infinite regress problem by ensuring that at some point, the regress will "bottom out" at a primitive actor.

However, the introduction of primitive actors means that a pure actor language is not necessarily more conceptually economical than an impure actor language: Primitive actors have special properties which distinguish them from other actors, so instead of having a single concept of actors, this model requires two separate concepts: Primitive actors, and all other actors. Furthermore, the infinite regress problem means that the concept of message passing in pure actor languages can be somewhat hard to understand, and to explain.

Everything is Not an Actor

However, there is a more fundamental, philosophical problem with pure actor languages: Everything is not an actor. Like objects in pure object-oriented languages, actors are entities that have identity and behaviour. However, not all phenomena lend themselves well to being modelled in terms of identity and behaviour. For instance, consider integer arithmetic. In a purely actor-based view of the world, we can view an integer as an actor (possibly a primitive actor) which has defined reactions to the `+`, `-`, `*` and `/` messages it receives. However, the concept of identity brings forth major philosophical problems. An actor retains its identity over time, despite any behaviour changes. This is a powerful concept for modelling real-world, *physical* phenomena in both object-oriented and actor-based programming: Because of identity, a car that has one of its tires changed is still considered the same car. However, *abstract* entities such as integers break the model: There is no way to change the behaviour of the number 42 and still have it retain its "42-ness". Its identity *is* its value[3].

Thus, it may be the case that a pure actor language necessarily has to seriously violate our understanding of some phenomena, such as integers. We generally don't think of

[3]A similar criticism of pure object-oriented languages has been put forth in *Everything is Not an Object*[Pai98].

integers as autonomous entities which react to messages, or entities that have existence in time. They are abstract, mathematical concepts which can be used to reason about the physical universe, but they do not themselves exist as physical entities. An impure actor language is able to reflect that philosophy directly, whereas a pure actor language must model abstract, non-physical concepts in terms of actors.

For these two reasons, and because of the emphasis on simplicity, MERLIN is an impure actor language.

3.3.3 Actors and Pattern Matching

A concept in MERLIN that is much easier to understand because messages are not actors is pattern matching. Pattern matching is a mechanism most often seen in functional and logic languages, and can often be used to express programs in a very concise style. In MERLIN, actors perform pattern matching on their message queues, such that the actor will respond to any message in the queue that matches any of the patterns specified in the current script of the actor[4]. It is important to point out that pattern matching is applied to the entire queue, not just the last message received. This means that when an actor changes behaviour, it may react to any number of "old" messages that the patterns of the new behaviour can match. In section 3.2.6, we saw an example of how this can be very useful to enforce an ordering in messages.

Limitations of Guard Expressions

MERLIN pattern matching allows the use of guard expressions to refine matches – but as briefly mentioned in section 3.2, guard expressions are composed of only a limited subset of MERLIN expressions.

The reason for this limitation is because of the concern that if guards could execute arbitrary statements, they would violate the reactivity precept. This is because of the separation of *message delivery* and *message acceptance* in the actor model, and in MERLIN: When matching a particular message against a set of patterns, many guards are potentially invoked before the message is either accepted or buffered. Since guard expressions are evaluated *before* the message is accepted, they are also carried out before an event is initiated. Without the aforementioned limitiations in guard expressions, an actor would thus be able to change behaviour or interact with other actors *without accepting a message.* This violates the reactivity precept, and would make MERLIN programs very difficult to understand. Furthermore, fully expressive guards could produce errors that would be *very* difficult to locate[5]

[4]This use of pattern matching is inspired by the concurrent functional language ERLANG, which uses pattern matching for a similar purpose.

[5]In fact, all operations with observable side effects are hazardous in guard expressions, for precisely this reason. However, since all primitive data values in MERLIN are immutable, this is only an issue with actors.

Thus, guard expressions are limited to only performing actions that are not directly observable – they may not interact with any other actors, and they may not cause behaviour changes.

Tuples

Pattern matching is also why Merlin has tuples as a primitive datatype. Although it is possible to represent any data structure using actors[HB77], the combination of pattern matching and actor-based data structures has several problems. If there is no primitive for representing collections, then all messages either consist of a single primitive data value, or a single actor. This reduces the utility of pattern matching, and since the aforementioned limitations on guard expressions imply that guard expressions cannot send messages to actors, it would be impossible to match against any kind of data structure. For this reason, Merlin provides tuples as a primitive datatype.

3.3.4 The Representation of Actors

Since actors are a fundamental concept in Merlin, their representation warrants some attention. Two major questions came up while developing the Merlin actor concept: How messages are received, and how the local state is represented.

Explicit vs. Implicit Receive

One issue, which may at first seem rather trivial, is whether or not message reception is implicit, or if there is an explicit `receive` construct. In Merlin, message reception is entirely implicit: An actor begins receiving messages after it has finished processing the last message it received. In other message-passing languages, such as Erlang and the Scala actor system, an explicit receive command is used to place an actor in a receiving state. In this model, the control dispatch on message reception is given in the receive construct. For example, consider the Merlin actor given in the buffering example:

```
seq : [[(1, msg) | tell console "First step"
                   become [[(2, msg) | tell console "Second step"
                                       become seq ()]]]]
```

In a language with explicit reception, this somewhat convoluted program can be written in a style which more clearly shows the *sequence* between the two distinct behaviours of that actor. In the explicit reception language Erlang, this program could be written thus:

```
seq() ->
    receive
      {1, msg} -> console ! "First step"
    end
    receive
      {2, msg} -> console ! "Second step"
    end
    seq();
```

Explicit reception can thus allow some protocols for actor communication to be specified more clearly. This example also shows another important implication of explicit reception: In the explicit reception model, behaviour change becomes partially or fully implicit. An actor can change behaviour inline, simply by "falling through" one `receive` to another. It is possible to entirely remove the become construct from such a language.

However, the approach also increases the overall complexity of the programming model. The main contributing factor to this increase in complexity is that it requires some form of explicit control flow. In languages with explicit reception, some method needs to be in place to allow the actor to enclose its receive mechanism in some form of loop. In SCALA, this is usually handled using traditional imperative looping mechanisms such as while loops, whereas ERLANG, a functional language, uses tail recursion. From a simplicity perspective, there are drawbacks to both approaches.

The SCALA approach requires the language to include more concepts. First, the language needs to provide some form of explicit iteration constructs. Second, if actors need to support complex protocols, then constructs to break out of loops (possibly even multiple nested loops) are required. In MERLIN, this situation is handled simply with become and implicit reception.

The ERLANG-style approach requires the programmer to make sure that all actor process specifications are properly tail-recursive, or risk having actors which monotonically increase in memory usage over time[6]. Having to guarantee tail-recursiveness adds some "conceptual overhead" to the model, however, since the programmer needs to be aware of another possible source of error.

Thus, both of these approaches require that some form of iteration or recursion facility is available. However, if it is possible to specify iteration or recursion which does *not* contain a `receive`, then the event atomicity precept is violated. Restricting iteration or recursion to only work with `receive`s within them may appear just as "strange" as simply not having any built-in iteration or recursion mechanisms. MERLIN is thus based on implicit reception, due to the consideration of concept economy.

[6]It is possible to detect this error statically. The same mechanisms that are used to optimize tail calls in functional language compilers[App98] can be used to detect non-tail-recursive actor specifications.

Actor State

Another important consideration about the representation of actors concerns their local memories. There are two basic ways to represent memory in actors: They can use *parameters* (which is how memory is represented in MERLIN), or they can have *persistent local variables*. There are benefits and drawbacks to both models.

With parameters, all local memory in actors is explicit and syntactically evident. Every behaviour change includes a specification of *all* replacement parameter values. This makes it easy to spot specific memory modifications in the code of a program. However, this also means that there is no way for any individual actor to maintain memory outside of the parameters for its behaviour. This means that if an actor has to retain memory over a behaviour change from a script of arity n to one of arity $m < n$, then it must either package some of its old parameters into a tuple, or send them to another actor for storage (in which case it must still be aware of the address of the storage actor). This may seem quite cumbersome.

The parameter-based view of local actor memory is based on a basically *function-oriented* view of actors. From this perspective, an actor is viewed as a function which maps pairs of behaviours and incoming messages to triples of new behaviours, sets of outgoing messages, and sets of new actors[Agh85]. However, we could also have taken a more *object-oriented* view of actors. In this view, an actor is not a function, but a *data structure* which has been augmented with a thread, and which may contain executable code in its fields. This view of actors is implemented in the IO language, a prototype-based object-oriented language with actor-based concurrency. In this perspective, local memory is simply contained in slots of the actor data structure.

This leads to an actor memory model based on persistent local variables. The data structure works as an environment, in which bindings of field names to values can be stored. In such a model, behaviour changes can have a finer granularity than in the parameter-based model: A behaviour change consists of replacing a set of bindings in the data structure, using assignment. However, this model also raises a number of new issues. For one, when behaviour changes are used to model complex protocols, the coarser granularity implied by become and parameters may actually lead to clearer programs, because an entire set of receptors and parameters can be exchanged for another. Furthermore, the data structure model allows actors to have *stale memory*, slots which are left over from earlier behaviours, but which will never be used again. Since the binding is still stored in a slot within the actor, and since actors may change behaviour, there will often be no way to prove whether it will ever be useful again, and thus it cannot be garbage collected. Finally, memory modifications may be scattered all over the text of a program, which may reduce the readability of programs in this model

Neither model conflicts with any of the core precepts, so both could have been used. The parameter-based mechanism was chosen primarily because it appears that the drawbacks of that model are less severe than those of the variable-based mechanism, but good implementations of both models exist (MERLIN and ERLANG are examples of the former model, IO and SCALA are examples of the latter), and it is a matter of controversy whether one model can be said to be universally "better".

3.3.5 Event Atomicity

An interesting and conceptually appealing property of MERLIN programs is that individual events are atomic, and are processed in finite time. When an actor accepts a message, it is *guaranteed* that at some point in the future, that actor will finish processing it, and enter a state in which it awaits more messages.

The event atomicity property of MERLIN implies that classical deadlocks, in which a set of blocking calls prevent each other from ever finishing, do not occur in MERLIN programs. Like in the pure actor model, the analogous situation of *actor deadlock* can arise – and, precisely because of event atomicity, we can understand this phenomenon in MERLIN using the same means as those described for pure actors in section 2.3.

This would not be possible in a language which did not guarantee event atomicity, because the two properties of *passivity* and *insensitivity* would no longer be necessary and sufficient for the deadlock behaviour to occur. Specifically, the lack of event atomicity would imply that a deadlock situation can occur even in sets of actors in which one or more of the actors are *not* passive: If an actor is stuck in an infinite event, then it is possible that that actor never sends out more messages, even though it is not passive.

Thus, the property of event atomicity makes the phenomenon of actor deadlock much less complicated to reason about.

The Price of Event Atomicity

A concept that is nearly ubiquitous in sequential programming languages is the procedure. Procedures are abstractions over sequences of computational steps, and they are powerful, expressive concepts: Recursive procedures allow very concise solutions to many computational problems[AS96]. It would seem desirable for an actor language like MERLIN to have a similar concept: With procedures, individual MERLIN actors could abstract away sequences of primitive operations, which could make for shorter, more concise programs.

Unfortunately, recursive procedures necessarily violate the event atomicity precept. If recursive procedures are allowed, then it is possible for an actor to enter an infinite recursion. In that case, the actor will never finish the event that initiated the recursion, and it will never enter a state where it awaits more messages.

Non-recursive procedures do not have this problem. However, using only non-recursive procedures means that a procedure could not call *any* other procedure, since event atomicity is violated by any form of infinite recursion, including infinite mutual recursion. Thus, for event atomicity to be preserved, only an extremely limited form of procedures could be used. However, such limited procedures would not add substantially to the expressivity of MERLIN.

Therefore, to allow event atomicity, MERLIN has *no concept of procedures*. Whether or not this tradeoff is "worth it" is debatable, but it was made due to the focus of MERLIN: First and foremost, MERLIN is a language meant for studying the linguistic implications of the actor model, and the property of event atomicity is a distinguishing characteristic of the actor model.

3.3.6 On Bidirectional Asynchronous Communication

A very common pattern of communication in many computer programs is bidirectional communication. Familiar examples of bidirectional communication flow in sequential programming languages include method calls in object-oriented programming languages, as well as function calls in functional languages.

Unfortunately, the asynchronous communication mechanism of the actor model does not, and cannot, support bidirectionality as a primitive, atomic operation. By definition, sending an asynchronous message does not cause the sender to wait for a reply or even an acknowledgement. As such, bidirectional communication in asynchronous messaging systems is necessarily a compound action, consisting of at least two separate asynchronous message transmissions.

One abstraction over such a message pattern is *ask expressions*[Agh85, Man88, HA92], which have been used in the ACORE and HAL languages[7]. Ask expressions allow a programmer to write expressions which evaluate to the reply of some other actor, similar to how method calls in object-oriented programming languages work. This can increase the succinctness and clarity of some programs, compared to explicitly writing the two steps of the communication.

However, `ask` has some quite unfortunate properties as well. Firstly, `ask`, as described in *Acore: The Design of a Core Actor Language and its Compiler*[Man88], has context-sensitive semantics which is not entirely straightforward to understand. There are two distinct contexts in which an `ask` can occur:

Behaviour-independent : If no `become` statement is executed after an `ask`, then a new, anonymous actor is spawned to handle the actor's actions after the `ask` – similar, in fact, to how the factorial example in section 3.2.7 worked. This new actor is instantiated with an automatically generated behaviour, which awaits the reply from the target of the `ask`, and carries out everything the original actor would have done with the reply. Such actors are frequently called *customers*[Agh85], or *continuation actors*[Hew76].

[7]In HAL, this mechanism is referred to as a "RPC Send", although it is similar to the `ask` expressions of the older language ACORE.

Behaviour-dependent : On the other hand, when there is a subsequent `become` statement, the above mechanism will not work – it would be the customer that changed behaviour at the end, not the original actor. In this case, a new behaviour is generated like in the behaviour-independent scenario, but instead of `spawn`ing a new actor with that behaviour, the original actor `become`s the continuation behaviour. Provided that the target of the `ask` sends a reply, this is safe, since the subsequent `become` guarantees that the actor will eventually change state away from the continuation behaviour. While in the continuation behaviour, the actor *only* awaits the reply, and will not react to any other messages.

The only one of the two possible `ask` variants which could, in theory, be used in all cases is the behaviour-dependent one. Unfortunately, this would effectively remove all advantages of asynchronous communication when `ask` is used: After issuing an `ask`, an actor changes to the insensitive behaviour and awaits a reply. One problem with this is that it reduces concurrency in the system, but worse, because of the insensitivity, it also introduces the possibility for deadlock.

The behaviour-insensitive `ask` variant addresses this problem. When the behaviour of the actor cannot change during the processing of the event that issues the `ask`, then it is safe to accept another message while the `ask` is being processed. This situation is handled by spawning a continuation actor, which processes the rest of the event after the `ask`.

Recursion Datalock

One form of datalock that can arise in behaviour-dependent `ask` expressions is *recursion deadlock*. A similar phenomenon is known in synchronous communication systems, and has been described in section 2.2 of this report. A recursive, behaviour-dependent `ask` satisfies the preconditions for actor deadlock, where the defining characteristic for recursion deadlock is that the configuration A consists of only a single actor. Recall that an actor behaviour change affects how it accepts *subsequent* messages. This means that the change to the insensitive behaviour happens *before* the actor can process its recursive `ask`, which then means that the message issued by the `ask` will be buffered. Once the message is buffered, our configuration of one is *silent*. Because it changed to the insensitive behaviour, it is also *insensitive*. And because it is awaiting its own reply, it is *passive*.

`Ask` expressions *can* maintain event atomicity, but their introduction into MERLIN would mean that events are no longer syntactically evident: Any `ask` terminates the current event, either by creating a new actor and awaiting more messages, or by changing behaviour into an actor which awaits the reply message. For this reason, and because of the high emphasis placed on simplicity, MERLIN does not include a bidirectional communication primitive.

	Merlin	Erlang	Scala	Io
Event Atomicity	Yes	No	No	No
Reception Model	Implicit	Explicit	Explicit	Implicit
Pattern Matching	Yes	Yes	Optional	No
Unrecognized Messages	Buffered	Buffered	Buffered	Unbuffered
Memory Model	Behaviours	Func. Parameters	Variables	Slots
Bidirectional Messages	No	No	!? Operator	Futures
Sequential Component	Limited Imperative	Functional	Multiparadigm	OO

Table 3.2: Comparison of Modern Actor Programming Systems

3.4 Evaluation of the Merlin Language Design

As mentioned in the introduction, the purpose of the Merlin language design was to undertake an exploration of the language design space around the actor model, with an emphasis of investigating whether the actor paradigm could be used to enable a simpler, more comprehensible style of programming for concurrent programs. For this purpose, the Merlin language was developed. In this section, we will see how Merlin relates to other actor-based programming systems, and consider whether or not Merlin meets its own requirements.

3.4.1 Merlin and Other Modern Actor-Based Languages

In recent years, actor-based programming has seen a new rise in popularity, possibly because of the increasing importance of concurrent, parallel and distributed programming systems. Several actor-based programming systems have been developed in recent years, by a very varied group of developers, and for many different purposes. Table 3.2 summarizes some of the differences and similarities between Merlin and three other modern actor systems: Erlang, Scala and Io.

In section 3.3, we saw the rationales for how the Merlin design considered each of the language design decisions represented in Table 3.2. Many of the Merlin design choices stemmed from the guiding requirement of simplicity, and the fact that Merlin is intended as an exploratory "toy language", rather than as a true application development language. It seems appropriate to consider the Merlin design in poerspective, so we briefly outline which design considerations affected the *other* modern actor languages, and how the actor model has been used to meet those considerations. We will consider the Erlang and Scala languages, since no literature about the design process of Io was available at the time this thesis was written[8].

[8]Most of the information about Io in this report comes from the language website, and from experimenting with programming in the language

Erlang

ERLANG is possibly the most famous language which uses actor-based concurrency semantics. Furthermore, the ERLANG language has an impressive body of industrial application experience behind it. It was designed primarily for high-availability telecommunications applications, and is used in some of the telecommunication switches manufactured by Ericsson[Arm03]. ERLANG is conceptually a layered design[Roy06]: The sequential layer implements a small functional language, and the concurrent layer implements an actor-like message-passing concurrent language.

The primary concern in the design of ERLANG [Arm03] was high availability, manifested in the ability to introduce new code at runtime ("hot loading"), and the ability to gracefully cope with errors. Actor-based programming provides a number of important benefits in these two areas. Hot loading benefits greatly from the memory and program encapsulation of actors, as well as the message buffering: In terms of the actor model itself, we can understand the phenomenon of hot loading simply as an actor changing its behaviour. While that actor is being reconfigured, it can buffer its incoming messages until it has had its new script installed and is ready to process them.

Error handling in actor systems can also be understood in terms of the model itself. In the application domain of ERLANG, many of the errors that arise are typical of the distributed computing field[CDK94], such as messages arriving out of order, or individual nodes in a system crashing. We have already seen how the combination of behaviour changes and buffering can be used to handle messages arriving in a wrong order. However, crashes violate the finite nature of events in the pure actor model, since an actor can crash during the processing of a message. When an actor is *known* to have crashed, we can model a crashed node to have become a *sink*, which drops all further messages.

In ERLANG, crash situations can be handled using timeouts: When other actors do not get any observable reaction from the crashed node for some span of time, they consider it crashed, and may notify other actors of the crash. This allows ERLANG systems to reconfigure themselves to work around crashed nodes. MERLIN would not be particularly applicable for this application domain, because it does not support timeouts, and because the event atomicity precept prevents it from modelling crashed nodes.

Scala

The SCALA language was designed primarily for scalability[HO07]. Therefore, for a concurrency model to be consistent with the purposes of SCALA, it must be able to scale in *both* directions: It should be possible to model both very large-scale concurrent programs running on powerful networked computers, as well as small-scale concurrency suitable for embedded devices. Since SCALA is implemented on the Java Virtual Machine, it must implement its concurrency support in terms of the primitives provided by that platform.

Unfortunately, spawning threads is somewhat expensive on the JVM[HO07], and large-scale concurrency on JVM-based systems thus often have to rely on other programming models. A common pattern is *event-driven programming*, which unfortunately suffers from a phenomenon called "control inversion", which means that event-based systems model control flow in a *producer-centric* manner: It is not the component which *needs* information that initiates communication, it is the one which can *provide* it. This can give event-based programs a quite convoluted control flow[Fuc95]. However, actors provide a mechanism for encapsulating event loops in a linguistic construct that is much easier to understand, and the Scala actor system uses actors for precisely this purpose: To provide an abstraction over event loops and thus allow concurrent programs to scale to larger sizes than thread-based ones.

However, Scala actors are not "pure" actors: For instance, they can access shared memory[Hal06], even if that programming practice is not encouraged. Because the Scala runtime does not enforce pure message-passing semantics, its actor implementation can be very lightweight and efficient. Chapter 4 demonstrates an implementation of Merlin, which highlights some of the implementation problems in enforcing pure message-passing semantics.

3.4.2 Revisiting the Requirements

In the beginning of this chapter, we identified simplicity as a very important property of concurrent languages: Because of the inherent complexity of concurrency, it is of vital importance that a concurrent programming language adds only minimally to the complexity of programs. We identified the property of simplicity as one of *economy of thought*, and saw two perspectives on achieving simplicity in a programming language: The principle of *consistency*, and the principle of *abstraction*.

One method for achieving consistency in a programming language is by developing a set of "natural laws" for programs in the language[Mey00], and assuring that no language construct is in violation of these laws. In Merlin, we gave eight fundamental precepts, and took care to exclude constructs which would violate them. If we define an inconsistency as a language construct which violates one of the fundamental precepts, then we can indeed say that Merlin is a consistent language. However, it is reasonable to ask whether the precepts themselves follow the design philosophy and requirements we developed. Table 3.4.2 summarizes the interactions between the precepts and the design requirements.

Simplicity

The programming model of Merlin is quite conceptually simple, in the sense that it provides only a few simple constructs to build programs from. Although Merlin is not a pure actor language, we saw in section 3.3.2) that pure actor languages are not

	Simplicity	Expressivity	Composability	Static Analysis	Low Overhead
Actor-Based	+		+		
Message Passing	+	+	+	+	-
Asynchrony		+	+		-
Concurrency	-	+			
Reactivity	+			+	
Event Atomicity	+	-		+	
Pattern Matching		+			-
Buffering		+			-

Table 3.3: Conceptual Design Tradeoffs in Merlin

necessarily simpler than impure ones: They have to introduce different kinds of actors, and understanding message passing is somewhat complicated in such languages. Merlin solved this problem by simply introducing primitive, non-actor-based values. Apart from the primitive data types, all of Merlin can be understood in terms of actors and messages. Merlin simplifies reasoning about programs because of two assumptions that can be made about Merlin programs: Actors are purely reactive, with atomic events.

Expressivity

If there is one barrier to truly high expressivity in Merlin, it is the combination of event atomicity and the lack of a bidirectional communication mechanism. Even in the simple factorial example in section 3.2.7, we saw that a Merlin programmer has to explicitly manage the control and data otherwise implemented by call stacks in more conventional languages. A bidirectional communication primitive could remove some of this extra, explicit information from programs, but unfortunately, bidirectional communication mechanisms in concurrent programming are anything but simple. However, Merlin is not a decidedly inexpressive language: Message-passing concurrency can generally be quite concise, at least compared to the syntactic clutter often introduced by lock variables. The asynchronous messaging system allows *some* programs to be expressed using recursive communication. Finally, pattern matching proved quite expressive: expressive: In all the small examples, not a single explicit `if` statement had to be written.

Composability

In several of the simple examples, we saw how Merlin actors can be *proxies* for each other. This provides a quite elegant composition mechanism, since an actor in a chain of proxies does not need to know anything about any other actor than its own immediate successor. If such actors designed such that they can be instructed by other actors to change their target, then this form of composition can even evolve dynamically in a running system.

Static Analysis

Because Merlin uses dynamic types, it is not particularly amenable to static analysis. However, we noted in section 2.2 that reasoning on causality chains in message-passing systems is generally easier than reasoning in shared-memory concurrent systems. The reactivity and event atomicity precepts further assists in the analysis of Merlin programs. However, it should be noted that static analysis was not a concern in this project.

Low Overhead

We investigate some of the implementation challenges of Merlin further in chapter 4. However, at this point it is possible to make a few statements about the efficiency of a Merlin interpreter in both time and space: *Purely* message passing interaction will be expensive, since it implies that name bindings must be implemented using copying, rather than referencing, in some cases. Pattern matching is more complicated and less efficient than simple conditional statements. Finally, buffering (including the buffering implied by asynchronous messaging) will increase the memory load of a Merlin interpreter. As above, it is worth noting that this project did not consider efficiency a high priority.

3.4.3 Final Remarks on the Merlin Design

One of the language precepts in particular, that of event atomicity, proved to be quite conceptually costly. While Merlin programs have the conceptually appealing property that deadlock-like behaviour only occurs when the two preconditions defined in sectction 2.3 are present, it is worth asking whether this property is worth the price. And precisely because all language features are consistent with that property, Merlin does not have any recursive procedures, or iteration constructs. While Merlin programs can still perform recursive computations (as we saw in the factorial example), they have to do so in a quite strange style. It is not clear to which degree this "strangeness" seriously hampers the comprehensibility of Merlin programs, and to which degree it is just an aspect of programming in an unfamiliar programming paradigm.

The concurrency and coordination examples in the Merlin introduction showed an interesting aspect of Merlin: Rather than being details of implementation (such as acquiring and releasing locks in order), proper synchronization in Merlin programs is promoted to the program design level. This implies that Merlin does, in fact, enable a higher level of abstraction when thinking about concurrency-related problems.

3.5 Summary

This chapter documented the design process of the Merlin language, and gave a brief introduction to the language itself. We can summarize the design process in the following four steps:

1. A general *statement of philosophy* for Merlin was developed.
2. The *qualities that should be encouraged in programs* were considered, and from the combination of these and the aforementioned statment of philosophy, a prioritized list of design requirements was assembled.
3. A set of guiding "natural laws" for Merlin programs were developed, in the form of eight core precepts.
4. Several language constructs were proposed, and judged in terms of the philosophy, the requirements, and an analysis of whether they would violate any of the core precepts.

We showed how this process led to a quite simple actor-based language, and contrasted its development with two other actor-based languages, Erlang and Scala. However, one of the eight core precepts, that of event atomicity, proved to be very conceptually costly, in that it required us to discard some linguistic concepts that could improve the expressivity and applicability of the Merlin language.

Recommended Literature

Programming language design is an active and exciting area of computer science research, dealing with a broad range of philosophical, theoretical and practical problems. Examples of good textbooks dealing with this subject include *Programming Language Pragmatics*, *Essentials of Programming Languages* and *Concepts of Programming Languages*.

A very good article dealing with the philosophical principles of programming language design is Bertrand Meyer's *Principles of Language Design and Evolution*, which in turn is inspired by C.A.R. Hoare's classic *Hints on Programming Language Design. The Next 700 Programming Languages*, by Peter Landin, is also a quite enlightening read.

4. A Simple Merlin Interpreter

"When I use a word," Humpty Dumpty said, in a rather scornful tone," it means just what I choose it to mean, neither more nor less."
– "Through the Looking Glass", Lewis Carroll (1832 - 1898)

There is an evil tendency underlying all our technology - the tendency to do what is reasonable even when it isn't any good.
– Robert M. Pirsig

We will now study the semantics of Merlin, through the development of a simple Merlin interpreter. As such, the primary purpose of the interpreter developed in this chapter is *not* to give an *efficient* Merlin implementation, it is intended as a simple proof-of-concept implementation with two main purposes: Showing the semantics of Merlin on a form understandable to programmers[1], and demonstrating a reference implementation from which more sophisticated implementations can be built.

For this reason, the Merlin interpreter demonstrated here is not optimized for performance in neither space nor time. It is quite slow, and has a high memory consumption. It does not attempt to parallelize Merlin programs maximally. These omissions have been made to ensure that the interpreter is clearer and more comprehensible. Some possibilities for future improvements are discussed in the text of this chapter, as the interpreter is developed.

Only a subset of the full interpreter is described in detail. In particular, the small sequential sublanguage of Merlin is given little attention. We do not describe the intricacies of conditional statements, Boolean operators and other "standard fare" components of the interpreter. Readers interested in a complete implementation of Merlin can turn to Appendix B, which reproduces the full Scheme source code of the Merlin interpreter.

Chapter Structure

This chapter is divided into five parts, of which three are relatively small.

[1]In this case, a working knowledge of the PLT Scheme implementation is helpful. Scheme was chosen as the implementation language for this Merlin interpreter because Scheme is very well-suited for constructing, manipulating and processing abstract syntax trees, which greatly simplifies the development of interpreters. The semantics of Scheme itself has sometimes been given in the form of a metacircular interpreter, that is, an interpreter for Scheme in Scheme.

- *Overview of the Interpreter*: This section gives an introduction to the basic structure of the interpreter, and the components it consists of.

- *The Actor Directory*: The first of three sections which describe the internals of the interpreter, this chapter describes the central data structure used to keep track of the actors in the system.

- *The Actor Loop*: The largest section of this chapter, this introduces the main operating concepts used by all actors. It describes the implementation of the pattern matcher, the script execution engine, as well as some of the fundamental concepts of MERLIN.

- *Built-In Actors*: This small section describes the implementation of built-in actors.

- *Evaluation of the Interpreter*: The final section of this chapter describes some of the challenges and problems of the interpreter, and outlines some of the opportunities for improvement.

4.1 Overview of the Interpreter

The concurrent execution of MERLIN programs can be understood in terms of the interactions between actors, and as such, this introduction to the interpreter will mainly be concerned with the realization of actors. There are two primary components to the MERLIN actor engine: The *actor directory* and the *actor loop*. The actor directory is a global data structure which maintains information about all actors in the system, in particular a mapping of addresses to actual communication channels. The actor loop contains the kernel of behaviour shared universally by *all* actors: It accepts messages, executes actor scripts, keeps track of parameters and handles behaviour changes.

The actor loop, in turn, makes use of a range of procedures to implement the functionality of MERLIN actors. In particular, it depends on a *pattern matcher*, a *script processor* and an *environment model* for its work. The script processor gives implementations of the statements and operators which make up the text of MERLIN programs. The interpreter is completed by a set of *built-in actors.*

An important point about the interpreter is that it executes MERLIN programs in an *abstract syntax* form, given in a fully parenthesized prefix notation called *S-expressions.* The use of an abstract notation greatly simplifies the internal workings of the interpreter, and S-expressions are the natural form of data structuring in SCHEME, the implementation language of the interpreter. While a lexer and parser transforming the concrete MERLIN syntax used in the rest of this thesis could have been written, it was felt that this would not contribute substantially to the focus of this project.

4.2 The Actor Directory

The actor directory is a global data structure which keeps track of all actors in a running MERLIN program. It defines a mapping between unique actor addresses and their associated communication channels, and supports two basic operations: `Directory-lookup` and `directory-register`.

There are two important aspects to the identity of an actor: It must be *unique*, and it must serve as a *reference* to the actor. For the purposes of the reference implementation, we can ensure these two properties by representing the address of an actor as a key into a global hash table.

Here is the definition of the actor directory:

```
(define directory (make-hash-table))
(define (directory-lookup address) (hash-table-get directory address))
(define (actor-port addr) (car (directory-lookup addr)))
(define (directory-register address port control)
  (hash-table-put! directory address (list port control)))
```

The addresses are simply represented as integers. The runtime keeps track of the maximum address registered, and registering a new address increments the maximum address. This address generator is protected by a mutex, to prevent race conditions when multiple actors spawn new actors concurrently:

```
(define max-address 0)
(define address-sema (make-semaphore 1))
(define (make-unique-address)
  (semaphore-wait address-sema)
  (set! max-address (+ max-address 1))
  (semaphore-post address-sema)
  max-address)
```

This scheme works for demonstration purposes, but is not very suitable for use in a more sophisticated implementation, since the address integer of the last registered actor is monotonically increasing. This means that in a long-running system, such as a program running as a server, it will eventually become a very large number. In a SCHEME-based implementation, this means that the memory consumption of addresses increases over time. However, in an implementation in a language like C which has bounded integer sizes, it will eventually cause a system crash. These problems can be remedied by using some other type than integers for representing addresses, and by reusing addresses from actors that have been reclaimed by a garbage collector.

4.3 The Actor Loop

The core of any individual actor in a MERLIN program[2] is the actor loop. The general behaviour of the actor loop is quite straightforward:

1. The first action of the actor loop is to wait for messages, on the asynchronous communication channel of the actor. As described in our overview of asynchronous communication, message reception in this model is a blocking operation.
2. Once a message arrives, it is tested against the set of receptors defined for the actor.
3. If a match is found, the associated reaction is executed. The actor keeps track of its state throughout the execution of the reaction, in order to detect whether it should change behaviour, after control returns to the actor loop.
4. The actor uses any information obtained during the execution to determine the behaviour script and local values to use for its next execution.
5. If the actor changes behaviour, then it resends all its buffered messages to itself, in case one of them can be recognized with the new behaviour.
6. If no match was found in step 3, then the message is appended to the buffer.
7. Finally, the actor loops back to step 1.

The actor loop itself is a recursive function with the following definition:

```
(define (actor-loop addr chan script locals buffer)
  (let* ((msg (async-channel-get chan))
         (receptor (select-receptor msg script locals))
         (state (cons addr #f))
         (result (if receptor (script-exec (recept-reaction receptor)
                                           (recept-env receptor)
                                           state)))
         (next (next-behaviour state locals))
         (new-script (if next (next-script next) script))
         (new-locals (if next (next-env next) locals))
         (new-buffer (cond ((and receptor next) (retry-buffer chan buffer) '())
                           (receptor buffer)
                           (else (cons msg buffer)))))
    (actor-loop addr chan new-script new-locals new-buffer)))
```

[2]The built-in actors are an exception. As we will see, MERLIN built-in actors use a different internal definition than the standard actor loop. This is because the built-in actors, by their nature, are written in SCHEME, not MERLIN.

As is evident, the actor loop makes use of a significant number of non-trivial components during its operation. We will describe each of the major components of the actor loop in turn.

4.3.1 Environments

The `locals` parameter of the actor loop contains its local memory, represented as an *environment.* Environments in the MERLIN interpreter are represented as lists of *frames*, each of which contains a list of names and a corresponding list of values. This representation allows environments to be easily extended, simply by augmenting the list with another frame. Performing a name lookup in an environment is done through recursive lookups in each frame. This implementation of environments is inspired from the environment model of the metacircular SCHEME interpreter given in *Structure and Interpretation of Computer Programs*[AS96]. The most important differences between MERLIN environments and the environment model of SCHEME interpreters are twofold: For one, because of the behaviour change model of state, MERLIN has no assignment statement, and thus does not need an analogy to `set!` in SCHEME, and secondly, there are situations in which MERLIN needs to perform deep copies of environments. We will return to the copying of environments later in this section.

Three fundamental operations are defined on environments: *Name lookups*, which return the value bound to some name in the environment, *definitions*, which create new bindings in the environment, and *extensions*, which generate a new environment, extended with another frame containing bindings.

As mentioned, environments are represented as lists of frames. The following procedures define the frame data structure:

```
(define (make-frame vars vals) (cons vars vals))
(define (frame-names frame) (car frame))
(define (frame-values frame) (cdr frame))
(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))
```

Similarly, these procedures define the composition of environment data structures:

```
(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define empty-environment '())
```

Using these data structures, we can now define the three fundamental operations on MERLIN environments. First, the name lookup procedure recursively scans an environment for a specified name. Unlike its SCHEME counterpart as described in [AS96], it does not signal an error if it encounters an unbound name – it just returns false. This is because it is used extensively by the pattern matcher, and in this context is expected to encounter many unbound names.

```
(define (name-lookup name env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? name (car vars)) (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env empty-environment)
      #f
      (let ((frame (first-frame env)))
        (scan (frame-names frame) (frame-values frame)))))
  (env-loop env))
```

The procedure to make a new definition scans through the environment, and changes a name binding if it already exists. If it doesn't, then the binding is placed in the first frame.

```
(define (define-name! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars) (add-binding! var val frame))
            ((eq? var (car vars)) (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-names frame) (frame-values frame)))
  env)
```

Finally, extending an environment adds a new frame to it, with its own name list and value list.

```
(define (extend-environment vars vals base)
  (if (= (length vars) (length vals))
    (cons (make-frame vars vals) base)))
```

4.3.2 The Pattern Matcher

The pattern matcher is an integral part of the MERLIN interpreter, due to the central role of pattern matching in message reception. It takes a pattern and a message, and recursively matches each element of the pattern against each element of the message. When the matcher successfully matches an element against the pattern, it creates an extended environment, in which the name of the pattern element is bound to the value of the message element. If the entire match succeeds, then the response script to the message is executed in the environment generated by the matcher.

Since the pattern matcher uses patterns defined in behaviours, it is illustrative to see an example of a Merlin behaviour in abstract representation before going through the workings of the pattern matcher. Here is a translation of the type-checking console proxy from section:

```
(define str-proxy
  (behaviour ()
    ((msg (where (string? msg))) (tell console msg))
    (msg (tell alert "Type error in str-proxy!"))))
```

The pattern matcher will attempt each pattern the list of receptors sequentially, until either finding a match or reaching the end of the list.

The two following procedures can be used to extract the pattern and reaction elements from the *first* receptor in a list:

```
(define (script-pattern script) (caar script))
(define (script-reaction script) (cadar script))
```

The pattern matcher itself is defined through two procedures, one responsible for scanning through the receptors and handling guard expressions, the other for doing the actual structural matching. We call these functions `select-receptor` and `match`. The intuition behind receptor selection can be described informally as follows:

1. Given a message, a script and an environment, `select-receptor` will first attempt to determine if there is a structural match, by calling `match`. It strips an eventual guard expression from the pattern first.
2. If a match can be made, then the resulting environment returned from `match` is used to evaluate the guard expression. A full match occurs when there is a structural match *and* the guard expression evaluates to anything other than false.
3. In case of a full match, `select-receptor` returns the environment generated by `match`. Otherwise, it calls itself with the rest of its receptor list.

Correspondingly, here is an informal description of the structural matching:

1. Given a pattern, an expression and an environment, the behaviour of `match` depends on the type of the pattern.
   - If the pattern is a *symbol*, and that symbol corresponds to a name defined in the environment, then return the current environment if the value of that name is equal to the value of the expression. Otherwise, return the special symbol `no-match`.

- If the pattern is a *symbol*, and that symbol is *not* defined in the environment, then return an extended environment with that name bound to the value of the expression.
- If both the pattern and the expression are *pairs*, recursively call `match` into those pairs.

Finally, here are the SCHEME definitions of these two procedures:

```
(define (select-receptor msg script env)
  (if (null? script) #f
    (let* ((matching (match (strip-guard (script-pattern script)) msg env))
           (guard-accept? (check-guard (script-pattern script) matching))
           (match? (and guard-accept? (not (eq? matching 'no-match)))))
      (if match?
        (cons (script-reaction script) matching)
        (select-receptor msg (cdr script) env)))))

(define (match pattern exp env)
  (cond ((eq? env 'no-match) 'no-match)
        ((equal? pattern exp) env)
        ((symbol? pattern)
         (if (name-lookup pattern env)
           (if (equal? (name-exp pattern env) exp) env 'no-match)
           (extend-environment (list pattern) (list exp) env)))
        ((and (pair? pattern) (pair? exp))
         (match (cdr pattern) (cdr exp)
                (match (car pattern) (car exp) env)))
        (else 'no-match)))
```

Guard Expressions

Some patterns may have *guard expressions*, which specify additional constraints for the match. Guard expressions follow the semantics of Boolean expressions shared by both MERLIN and SCHEME: Everything that is not explicitly `false` is considered true. The pattern matcher is designed such that guard evalution is performed by the receptor selector rather than the structural matcher, to avoid overly complicating the structural matcher.

The receptor selector uses four procedures to handle guard expressions. `Has-guard?` is a simple predicate which determines if a pattern has a guard expression or not, `guard` returns the guard expression of a pattern which has one, `strip-guard` removes a guard expression from a pattern, and `check-guard` evaluates a guard expression within an environment given by the structural matcher:

```
(define (has-guard? pattern)
  (and (pair? (last pattern)) (eq? (car (last pattern)) 'where)))
(define (guard pattern) (cadr (last pattern)))
(define (strip-guard pattern)
  (if (has-guard? pattern)
    (take pattern (- (length pattern) 1))
    pattern))
(define (check-guard pattern env)
  (if (has-guard? pattern)
    (guard-eval (guard pattern) env)
    #t))
```

`Guard-eval` evaluates a side-effect-free subset of MERLIN expressions.

4.3.3 The Script Processor

The script processor is an important component in the MERLIN interpreter. It is similar in concept to the eval function in SCHEME interpreters, with the important distinction that MERLIN scripts are always evaluated in the context of an actor. The MERLIN script processor takes an expression, an environment, and a special *state variable* as parameters:

```
(define (script-exec exp env state)
  (cond ((self-evaluating? exp) exp)
        ((and-exp? exp) (eval-and (vals (operands exp) env state) env))
        ((or-exp? exp) (eval-or (vals (operands exp) env state) env))
        ((self-reference? exp) (cons 'address (car state)))
        ((name? exp) (name-exp exp env))
        ((definition? exp) (eval-definition exp env state))
        ((if? exp) (eval-if exp env state))
        ((tell? exp) (eval-tell exp env state))
        ((spawn? exp) (eval-spawn exp env state))
        ((become? exp) (eval-become exp env state))
        ((behaviour-definition? exp)
         (make-behaviour (behaviour-parameters exp) (behaviour-body exp) env))
        ((begin? exp) (eval-sequence (begin-actions exp) env state))
        ((operation? exp)
         (apply-operation (script-exec (operator exp) env state)
                          (vals (operands exp) env state)))
        (else (error "SCRIPT-EXEC ERROR: " exp))))
```

The MERLIN script processor is a recursive procedure which performs a case analysis on the input expression given the input environment. Like a SCHEME evaluator, the script processor contains all the "special forms" of the language – in the case of MERLIN, all the statements[3].

[3]To simplify the interpreter, statements are considered expressions that have undefined return values. If a statement occurs in an expression context, where the value is required, the underlying SCHEME interpreter will signal a type error.

Expressions and Operators

Unlike SCHEME evaluators, the corresponding `apply` procedure (`apply-operation` in MERLIN) is very simple:

```
(define (apply-operation operator operands) (apply (car operator) operands))
```

This is because, unlike SCHEME, MERLIN does not have a notion of functions or user-definable operators.

All the operators for MERLIN expressions are defined in an association list, `operators`, which is installed into the environment of the top-level actor during the initialization of the interpreter. This association list maps operator names to their implementations, many of which are taken directly from the underlying SCHEME interpreter:

```
(list
    (list '+ +)
    (list '- -)
    (list '* *)
    (list '/ /)
    ...
    (list 'tuple? tuple?)
    (list 'behaviour? behaviour?)
    (list 'address? address?)
    (list 'is equal?)))
```

Only a subset of the operator list is reproduced here; the full list can be seen in Appendix B.

The Representation of Behaviours

The representation of behaviours is a rather complicated and somewhat tricky part of the MERLIN interpreter, because of the combination of static scope and pure message-passing concurrency semantics. Because of the static scope, behaviours need to contain a representation of the environment in which the behaviour was defined, and because of the purely message-passing concurrency semantics, it cannot simply use a reference to that environment, like SCHEME interpreters do. If a reference was used, then the actor that defined the behaviour could engage in a degenerate form of shared-state concurrency with all the actors spawned from that behaviour, by redefining values in its environment. This problem is solved by performing a deep copy of the environment in which a behaviour is created. The following procedures give a definition of the representation of behaviours in MERLIN programs:

```
(define (make-behaviour params script env)
  (list 'behaviour params script (deep-copy env)))
```

```
(define (behaviour-params bhv) (cadr bhv))
(define (behaviour-script bhv) (caddr bhv))
(define (behaviour-env bhv) (cadddr bhv))
```

The `make-behaviour` procedure works well when defining anonymous behaviours, but unfortunately, it does not allow self-references in behaviours. This is because the behaviour is (obviously) not yet defined at the time when it is defined, and thus the copied environment does not contain a reference to it. Since self-referential behaviours are a *very* common construct in MERLIN programs, occurring whenever an actor changes only the parameters of its current behaviour, some workaround for this problem is needed.

Since this is only a problem in the case of named behaviours, we solve this by introducing a special case in the procedure for name definitions. Although this is somewhat inelegant, it does solve the problem for simple self-reference:

```
(define (eval-definition exp env state)
  (if (and (pair? (definition-value exp))
           (eq? (car (definition-value exp)) 'behaviour))
    (begin
      (let ((new-env (deep-copy env)))
        (define-name! (definition-name exp)
                      (list 'behaviour
                            (behaviour-parameters (definition-value exp))
                            (behaviour-body (definition-value exp))
                            'this-environment)
                      new-env)
        (define-name! (definition-name exp)
                      (list 'behaviour
                            (behaviour-parameters (definition-value exp))
                            (behaviour-body (definition-value exp))
                            new-env)
                      env)))
    (begin
      (define-name! (definition-name exp)
                    (script-exec (definition-value exp) env state)
                    env))))
```

The environment copy stored in the behaviour is augmented with a reference to the behaviour itself, in which the environment is represented by the symbol `this-environment`. This is necessary to avoid introducing a cycle in the environment, which would otherwise cause an infinite loop in the next deep copy. The `this-environment` symbol is handled and expanded to the proper environment by the which is used by the main actor loop. This mechanism is described below, in the description of the behaviour change process.

Communication

As we have seen in the previous chapters, a defining trait of the actor model is its asynchronous communication model. In MERLIN, this is modelled using the `tell` primitive, defined in the interpreter by the following SCHEME procedure:

```
(define (eval-tell exp env state)
  (let* ((values (vals (operands exp) env state))
         (addr (cdar values))
         (msg (cadr values)))
    (async-channel-put (actor-port addr) msg)))
```

The `async-channel-put` procedure is defined in the PLT SCHEME `async-channel` module, which provides buffered, asynchronous communication channels similar to those used in the actor model. Recall that the `actor-port` procedure is used in the interface to the actor directory.

We already saw the receiving side of the communication, in the actor loop. The built-in buffers of the PLT SCHEME channels provide part of the functionality of MERLIN buffers: They ensure that an actor will receive messages, even when it is busy doing something else. The explicit buffer implemented by the actor loop is used to handle the remaining part of the buffer functionality defined by MERLIN: Buffering unrecognized messages.

Spawning New Actors

Another important aspect of actor semantics is that actors can create new actors. The `spawn` statement of MERLIN is used to create new actors, and is given by the following procedure:

```
(define (eval-spawn exp env state)
  (let* ((values (vals (operands exp) env state))
         (bhv (car values))
         (actuals (cadr values)))
    (cons 'address (new-actor bhv actuals))))
```

Like `tell`, this procedure needs to interact with the actor directory. This interaction is handled through the `new-actor` procedure, which is responsible for creating an address, channel and thread for the actor. It is given here, together with its helper procedures `actor-thread` and `initial-env`:

```
(define (initial-env bhv actuals)
  (extend-environment (behaviour-params bhv) actuals (behaviour-env bhv)))
(define (actor-thread addr port bhv actuals)
  (thread (lambda ()
            (actor-loop
              addr port (behaviour-script bhv) (initial-env bhv actuals) '()))))
```

```
(define (new-actor behaviour actuals)
  (let* ((addr (make-unique-address))
         (port (make-async-channel))
         (control (actor-thread addr port behaviour actuals)))
    (directory-register addr port control)
    addr))
```

Changing Behaviour

We have already seen some of the challenges of implementing the behaviour change model of state, in the description of the representation of behaviours in the MERLIN interpreter.

The behaviour-change interaction between the actor loop and the script processor in the MERLIN interpreter is done through a *state variable*, called `state` in both the actor loop, the script processor, and in all the procedures called by the script processor that require it. As we saw in the definition of the actor loop, it is a pair in which the first field contains the *actor address* (which is used for another purpose in the interpreter, namely allowing actors to access their own addresses using the `self` expression), and in which the second field is initialized to simply contain the Boolean false value. It is this second field which is used to contain behaviour change information.

The `eval-become` procedure implements behaviour changes in terms of the state variable:

```
(define (eval-become exp env state)
  (set-cdr! state (list (script-exec (become-params exp) env state)
                        (script-exec (become-script exp) env state))))
```

When the call from the actor loop to `script-eval` finishes, the formal parameters and behaviour definition (including the environment copy) will be stored in the state variable[4].

When control has returned to the actor loop, the actor uses the `next-behaviour` and `calculate-next-environment` procedures to determine its next script and local values, along with two data abstraction functions to access the state variable:

```
(define (next-behaviour state locals)
  (if (cdr state)
    (let* ((specified-env (behaviour-env (state-behaviour state)))
           (next-env (calculate-next-environment state locals))
           (next-script (behaviour-script (state-behaviour state))))
      (cons next-script next-env))
    #f))
```

[4]While programming by side effect is generally considered bad practice in SCHEME, the interpreter would be unnecessarily complicated if this was done without side effects, by passing the state along with the return value.

```
(define (calculate-next-environment state locals)
  (let* ((specified (behaviour-env (state-behaviour state)))
         (next-env (if (eq? specified 'this-environment) locals specified)))
    (extend-environment
      (behaviour-params (state-behaviour state))
      (state-actual-params state)
      (enclosing-environment next-env))))

(define (state-behaviour statevar) (caddr statevar))
(define (state-actual-params statevar) (cadr statevar))
```

The `calculate-next-environment` procedure handles the `this-environment` symbols, which we introduced earlier, by simply keeping the current environment. Also note that the parameters are installed in the first frame of the environment, allowing the parameter set to be changed simply by extending the enclosing environment with the new parameter bindings.

4.4 Built-in Actors

Finally, a MERLIN interpreter needs to provide implementations of the two built-in actors `void` and `console`. To install primitive actors, the interpreter uses the following procedure, analogous to the one used to spawn normal actors:

```
(define (register-primitive-actor func)
  (let* ((addr (make-unique-address))
         (chan (make-async-channel))
         (control (thread (lambda () (func chan)))))
    (directory-register addr chan control)
    addr))
```

This procedure simply generates the address, channel and control thread for an actor. Its single input parameter must be a recursive function of one argument, where the one argument will be used for the channel. The `void` and `console` functions themselves are quite straightforward:

```
(define (console-proc chan)
  (let* ((msg (async-channel-get chan)))
    (write msg) (newline)
    (console-proc chan)))

(define (void-proc chan)
  (let* ((msg (async-channel-get chan))) (void-proc chan)))

(define console-actor (register-primitive-actor console-proc))
(define void-actor (register-primitive-actor void-proc))
```

4.5 Evaluation of the Interpreter

The interpreter given here is very inefficient, in both space and time. However, it does give a fairly clear description of the semantics of the MERLIN language, and reveals some of the challenges in developing a higher-quality implementation of MERLIN.

In particular, there are obvious problems with the handling of the behaviour-change model of state in this interpreter. The deep copy of the entire environment in all behaviour definitions is extremely inefficient, and also introduces problems with self-referencing behaviours – problems which have rather complicated solutions. This is especially problematic because MERLIN programs treat behaviours as a first-class value, and allows them to be defined inline. This encourages defining many of them, which will drastically reduce program performance unless an optimization strategy for this problem is developed. For example, it may be feasible to only copy bindings of names that actually appear within the definition of a behaviour.

Another problem stems from the handling of message buffers. In MERLIN, buffered messages should be kept around indefinitely, because an actor may later change behaviour into one which may be capable of receiving the message. This presents two problems: For one, the interpreter given here takes the simplest route of handling this by simply having actors re-send their entire message buffers to themselves when changing behaviour, which is quite inefficient. Second, it is possible for an actor to cause an unbounded memory leak by sending masses of unrecognizable messages to itself, or to some other actor. This problem can be remedied in some cases: If an implementation can *prove* that a particular actor will never be able to assume a behaviour in which it can accept some message, then it is allowed to delete that message. However, this is not possible in all cases, so a MERLIN programmer should probably make it a habit to include a catch-all "receive and do nothing" receptor for behaviours that do not require the unlimited buffering facility.

4.5.1 Limitations

A notable omission of this interpreter is garbage collection. Because actors are never removed from the actor directory, the underlying SCHEME interpreter will always have a reference to them. This prevents them from being garbage collected by the underlying SCHEME interpreter. Since the MERLIN interpreter itself has no garbage collection, this means that programs in this interpreter have monotonically growing memory usage, and will eventually consume the memory of any host.

Due to the same problems that mandated the introduction of the `this-environment` construct in the representation of behaviours, this implementation of the MERLIN interpreter does not support forward references in behaviours. In other words, if behaviour A lexically precedes behaviour B in a MERLIN program text, then an actor with behaviour A cannot refer directly to behaviour B. It is still possible for an actor with behaviour

A to change to behaviour B, however. Because behaviours are first-class values, it is possible for some actor to send B to that actor, allowing it to change to it. This is quite cumbersome, though, and would not be acceptable in a production implementation.

This problem is analogous to the problem of supporting mutual recursion in procedures, and one solution proposal to this problem is given in Andrew Appel's *Modern Compiler Implementation in C*[App98]: The parser can translate lexically adjacent procedure definitions to a *list* of procedure definitions, in which forward references and mutual recursion is allowed. This method could be adapted to MERLIN behaviour definitions.

Another limitation is that this interpreter has very poor error reporting facilities, and generally falls back on the error reporting mechanisms of the underlying SCHEME interpreter when it encounters type and arity errors.

4.6 Summary

This chapter described the development of a simple, although limited and inefficient, abstract syntax interpreter for MERLIN. The interpreter highlighted some principles behind the operational realization of the semantics of MERLIN, defined as a collection of procedures in the SCHEME language. We saw some of the points where the MERLIN language presents implementation challenges, particularly in the behaviour-change model of state, and in the enforcement of pure message-passing semantics.

Recommended Literature

A textbook which, among many other things, gives a good treatment of interpreter construction is the excellent *Structure and Interpretation of Computer Programs*, by Harold Abelson, Gerald Sussman and Julie Sussman. This book introduces the art of interpreter construction through the development of a set of interpreters for variants of the SCHEME language, themselves written in SCHEME. This book had a large part in my decision to write the demonstration interpreter for MERLIN in SCHEME. Two other very good books on the subtler points of writing interpreters include *Essentials of Programming Languages*[FWH92] and *Lisp in Small Pieces*[Que96], both of which also use SCHEME as their implementation language.

Interpreters are not the only way to implement programming languages, of course. Efficient implementations of programming languages are generally written as compilers, either to machine code or to various bytecode-based virtual machines. A good introduction to compiler construction is given in Andrew Appel's famous "tiger book" series, *Modern Compiler Implementation*[App98], which is available in different versions with code given for C, ML and JAVA. Furthermore, no compiler bookshelf is complete without the "dragon book", *Compilers: Principles, Techniques and Tools*[ASU86], by Jeffrey Ullman, Alfred Aho and Ravi Sethi.

5. Conclusion

That is what learning is. You suddenly understand something you've understood all your life, but in a new way.
– Doris Lessing

In the introduction, we identified the core problem of this thesis as the question of which benefits and challenges the actor model presents as a basis for concurrent programming language design. We adopted the approach of developing a small, academic "toy language" to investigate this problem. The preceding chapters focused on the development of Merlin, a small concurrent language based on the actor model.

The first question we addressed was whether concurrent programming is *inherently* more complex and difficult than sequential programming. We used a small conceptual framework, consisting of *computational steps*, *executions*, *processes* and the *orderings* and *interleavings* between computational steps, to investigate the intuition that concurrent programming is naturally difficult. We found that the fundamental problem underlying this difficulty is that while a sequential program must only be correct for a single interleaving, a concurrent program must be correct for *all* possible interleavings.

Following this result, we investigated the concepts of concurrent programming models, which reduce the total set of possible interleavings, and which allow programmers to reason about the possible interleavings of executions. We reviewed several concurrency models, ranging from a very restrictive observationally deterministic model, in which only non-observable differences in interleavings between executions are possible, to fully expressive models based on two widely different core abstractions. We developed a taxonomy of concurrency models, allowing us to consider *why* the actor model is designed the way it is, with asynchrony, message-passing and nondeterminism at its core.

This put us in a position to undertake a more complete investigation of the actor model itself. We saw that one of the defining characteristics of actor systems is the very important point that *individual actors are strictly limited in computational power* – an individual actor, without use of the communication system, is not Turing-complete. We saw that this property has the quite interesting implication that deadlock in actor systems can be understood in terms of the two preconditions of *passivity* and *insensitivity.* This allows a general strategy for avoiding actor deadlock by avoiding, or restricting, the insensitivity condition. In general, a very appealing property of the actor-based paradigm is that many concurrency concerns are promoted to the program design level, rather than being viewed as matters of implementation detail.

Partial Conclusion: At this point, we can conclude that the actor model does, in fact, have two very desirable benefits for programming language design: It *simplifies reasoning* about concurrency problems, exemplified by the two simple preconditions for actor deadlock, and it allows a *high level of abstraction* for concurrency, which is precisely what is implied in the promotion of concurrency concerns to matters of program design.

The next part of the report concerned the design process of the Merlin programming language. We gave an overall design philosophy for the Merlin language, based on the guiding concepts of *simplicity*, *consistency* and *abstraction*, with the final goal of producing a language which has a *light mental footprint*. It was argued that careful attention to simplicity is *especially* important in concurrent languages, due to the higher inherent complexity of concurrent progamming. The design process of the Merlin language attempted to take this into consideration, by giving a set of eight *core precepts*, which were used to exclude proposed language features that would be inconsistent with the language as a whole.

An important core precept is *event atomicity*, which captures the aspect of actor computation that any event (defined as the interval between accepting a message and finishing its processing) may last for only a bounded interval of time. Unfortunately, while this precept allows the same simple reasoning about actor deadlock and similar phenomena as the pure actor model, it also places quite severe restrictions on which linguistic constructs can be supported within Merlin: Recursive procedures, for example, cannot be supported by actors in a language which guarantees event atomicity. It is unclear whether this tradeoff can be considered reasonable.

Another linguistic construct which was sacrificed for the reason of simplicity was *bidirectional communication*. We saw that bidirectional communication is *necessarily* a compound action in actor programs, and that some patterns of bidirectional communication (particularly recursion) satisfy both of the preconditions for actor deadlock. This appears to be a general problem for message-passing concurrency systems, since we have also seen that recursive communication leads to deadlock in synchronous message-passing systems. The asynchronous communication system of the actor model can remove the risk of deadlock in certain cases – but that this comes at the price of a highly context-sensitive semantics for bidirectional communication, which introduces a significant degree of additional complexity into actor programs. In particular, while it is possible to have bidirectional communication that does not violate event atomicity, such a communication scheme makes it difficult to read from a program what actually constitutes an event.

Unfortunately, the lack of recursive procedures *and* bidirectional communication primitives means that certain Merlin programs (particularly those that implement recursive computations) must be written in a somewhat “odd” style, reminiscent of Continuation-Passing Style, an intermediate representation used by some functional language compilers.

Partial Conclusion: Linguistic *benefits* of actor-based programming is that the desirable properties of the actor model itself can carry over to concrete computer programs, allowing programmers to reason at a high level about concurrency in their programs. This requires the programming model to be conceptually close to the actor model itself. The two primary *problems* include the necessarily complicated (and deadlock-prone) nature of bidirectional communication in actor programs, and the fact that useful linguistic constructs, like recursive procedures cannot be supported in actor languages without giving up some of the ability to reason about programs.

Finally, we studied the semantics of Merlin by implementing an interpreter for it. This interpreter highlighted some of the implementation challenges facing actor languages. One that stood out was that the interpreter *copies* all bindings of environments in some cases, to prevent a degenerate form of shared-state concurrency from occurring. This copying is not only highly inefficient, it also complicates the implementation of certain language features, such as first-class behaviour values and self-referential behaviours. Unfortunately, the presence of first-class behaviour values encourages their use, which only adds to the problem of inefficiency.

Final Conclusion: Actor-based programming languages have the very important benefits that their concurrent semantics is simpler to reason about, and that concurrency concerns are matters of high-level design in actor programs. This means that they can be very powerful tools for conceptualizing and thinking about concurrent programs. Unfortunately, the actor model also has its own drawbacks and challenges: Bidirectional communication is a complicated and possibly deadlock-prone action, and some of the linguistic constructs many programmers take for granted conflict with the nature of actors. Furthermore, the efficient implementation of actor languages entails some challenges, due to the combination of pure message passing and the behaviour-based model of state.

The abstractions provided by the actor model are very useful, however, in a world where the need for good concurrent programming practices is steadily increasing, and where it becomes ever more urgent for programmers to be able to *think* clearly about concurrency in programs. Despite its age and the implementation challenges, the actor model still has much to offer. However, to develop a concrete, practical programming paradigm from the model, more research is clearly needed.

6. Future Work

Prediction is very difficult, especially about the future.
– Niels Bohr (1885 - 1962)

The best way to predict the future is to invent it.
– Alan Kay

This, the final chapter of this thesis, gives some brief outlines of possibilities for future work with the Merlin language and the actor model.

6.1 Further Evolution of the Merlin Language

The Merlin language, as it is presented here, is an example of what is often called – affectionately or perjoratively – an academic toy language. These are languages which are useful for performing computer science experiments, such as exploring linguistic concepts and programming paradigms, but generally not suitable for production use. While this is well within the intentions and scope of this project, it is nonetheless interesting to consider whether and how Merlin can be adapted into a more productive, expressive language, which could be used in actual software projects.

6.1.1 Efficient Implementation

One of the main failings of the Merlin interpreter presented in this work is that it is very slow and memory-inefficient. There are several reasons for its poor performance:

- It is implemented as an interpreter running on top of another interpreter – which, itself, is not among the faster Scheme interpreters.
- It performs much needless work. For example, it performs unnecessary copying of environment frames.
- It does not implement garbage collection. The garbage collector of the underlying Scheme implementation does not work, since the actor directory maintains references to even garbage actors.
- It works by directly traversing and interpreting the abstract syntax trees of Merlin programs. This is not a particularly efficient method of interpretation; generally,

fast interpreters translate programs into more efficient representations before interpretation.

A major part of the inefficiency can be addressed by rewriting the interpreter in an efficient, compiled language, such as C or ML. Furthermore, rather than using a simple tree interpreter, a more efficient implementation can be implemented as a virtual machine and bytecode compiler. Developing an actor-based virtual machine presents an interesting and challenging future project.

Actor languages are not entirely straightforward to garbage collect, since a full actor garbage collector will essentially have to garbage collect *processes* as well as data. Some formal work in actor garbage collection has been done: In *Actor Systems for Real-Time Computation*[Bak78], Henry G. Baker develops an incremental, real-time garbage collector for an actor language. Implementing an actor garbage collection scheme can be another quite interesting and challenging future development for MERLIN.

6.1.2 Expressive Sequential Sublanguage

While MERLIN presents a useful model for writing concurrent, interacting and coordinating actors, its means for actually specifying the internal behaviour of these actors is quite weak. This weakness is partially deliberate, since much of it stems from the lack of procedures and functions. The lack of procedures and functions is intended to prevent recursion, which can be potentially infinite. In the case of infinite recursions, the event atomicity property is violated, which means that the normals means for reasoning about the phenomenon of deadlock in actor systems will no longer work. There are two basic options for further developing the sequential sublanguage of MERLIN: Exploring how far we can go without sacrificing event atomicity, or giving up event atomicity in the name of pragmatism. Both ideas have some merit.

In the first scenario, the task is to make the sequential sublanguage reasonably expressive using only constructs which are guaranteed to terminate. For example, various forms of bounded iterations could be included, such as list comprehensions. The exclusive use of bounded constructs means that the sequential sublanguage will remain Turing-incomplete, which in turn implies that many programs in such a language will *have* to use actors and communication to work. This is not necessarily a bad thing: Actors are, after all, a distinguishing characteristic of the MERLIN language.

Alternately, we can give up event atomicity and use a Turing-complete sequential sublanguage. In this model, some errors are possible that do not exist in the current MERLIN implementation, namely the error of *actor unresponsiveness*. An actor is unresponsive when it enters a state where it will never await further messages. It is impossible to detect this error, since successfully detecting it in the general case is equivalent to solving the Turing Halting Problem. In a MERLIN version with a Turing-complete sequential sublanguage, infinite loops would most likely be considered a programmer error just like they often are in sequential languages. A Turing-complete sequential sublanguage could take the form of any functional or imperative programming language.

The programming language design pattern of using several layers of language, in which one language layer implements a reasonably expressive sequential language and another provides concurrency, turns out to be quite successful. The E, Erlang and Oz languages are all based on this design, which has been investigated further in Peter Van Roy's *Convergence in Language Design: A Case of Lightning Striking Four Times In The Same Place*[Roy06].

6.1.3 Error Handling and Recovery

One particularly attractive trait of message-passing concurrent models is that they can be very resilient to error, because of the independence of the message-passing entities. If a message-passing component in a properly designed system fails, others can either reorganize themselves to take over its responsibility, or spawn a working replacement for it. This aspect of message-passing programs has been explored extensively in Joe Armstrong's Ph.D thesis *Making Reliable Distributed Systems in the Presence of Software Error*[Arm03], which demonstrates how the Erlang language realizes these advanced error handling mechanisms.

Merlin has almost completely ignored the issue of error management, but since it is a message-passing language in which actors are independent, it should be possible to implement advanced error management facilities in Merlin.

6.2 Further Work with the Actor Model

The actor model itself rests on a solid theoretical foundation, but theoretical work with the model is still being performed. Its founders are still active in the computer science research community, and still actively contribute to the model. However, as we have seen, the model is not without its problems. More research into the actor model can prove quite fruitful in the future, as the need for concurrent and parallel software is likely to keep increasing.

6.2.1 Bidirectional Actor Communication

A shortcoming of the actor model is the relative difficulty of implementing bidirectional communication between actors. The most common mechanism, the `ask` expression, has the somewhat confusing property of having heavily context-dependent semantics – a problem which is only compounded by the fact that careless use of `ask` may lead to deadlocks. Because of the context-dependent semantics, investigating a program which deadlocks due to a faulty `ask` is not straightforward, since the *context* of each `ask` must also be taken into consideration.

While bidirectional communication must necessarily be a composite action in an asynchronous communication model such as the one used by the actor model, it may be the case that less error-prone and semantically complicated means of bidirectional communication can be developed.

6.2.2 Handling Deadlock in Actor Systems

Because the actor model uses purely asynchronous communication, classic deadlock cannot occur in actor programs. However, we have seen that a similar situation *can* arise, in which an actor configuration is passive and insensitive , and that this situation can be detected.

Some means of preventing this situation have been developed, such as the *enabled-sets* of the ROSETTE language. These allow actors to selectively disable responses to particular messages, which can be used to prevent deadlock by removing the insensitivity condition. However, enabled-sets can be somewhat complicated to work with. This complexity may or may not be due to the inherent complexity of dealing with deadlock, but at any rate, it may prove interesting to further study the phenomenon of actor deadlocks, and mechanisms for avoiding, preventing, detecting and recovering from them.

Bibliography

[Agh85] Gul Agha. Actors: A model of concurrent computation in distributed systems. 1985.

[Agh86] Gul Agha. An overview of actor languages. *SIGPLAN Notices*, 1986.

[AH78] Giuseppe Attardi and Carl Hewitt. Specifying and proving properties of guardians for distributed systems. 1978.

[App98] Andrew W. Appel. *Modern Compiler Implementation in C.* Cambridge University Press, 1998.

[Arm03] Joe Armstrong. *Making reliable distributed systems in the presence of software errors.* PhD thesis, The Royal Institute of Technology Stockholm, 2003.

[AS96] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, 1996.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[AWV93] J. Armstrong, D. Williams, and R. Virding. *Concurrent Programming in Erlang.* Prentice Hall, 1993.

[Bak78] Henry G. Baker. *Actor Systems for Real-Time Computation.* PhD thesis, 1978.

[CDK94] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems Concepts and Design.* Addison Wesley, 1994.

[Cli81] William Clinger. *Foundation of Actor Semantics.* PhD thesis, 1981.

[Dek05] Steve Dekorte. Io: a small programming language. In Ralph Johnson and Richard P. Gabriel, editors, *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2004, San Diego, CA, USA*, pages 166–167. ACM, 2005.

[Dij68] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:1115–138, 1971.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[Eck98] Bruce Eckel. *Thinking in Java.* Prentice Hall, London, 1998.

[ELa] The E Language. http://www.erights.org/.

[Eng04] John English. *Introduction to Operating Systems: Behind the Desktop.* Palgrave, 2004.

[Fuc95] M. Fuchs. Escaping the event loop: an alternative control structure for multi-threaded GUIs. In Leonard J. Bass and Claus Unger, editors, *EHCI*, volume 45 of *IFIP Conference Proceedings*, pages 69–87. Chapman & Hall, 1995.

[FWH92] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* McGraw Hill, 1992.

[GH75] Irene Greif and Carl Hewitt. Actor semantics of planner-73. 1975.

[Gre75] Irene Greif. *Semantics of Communicating Parallel Processes.* PhD thesis, 1975.

[HA79] Carl Hewitt and Russell Atkinson. Specifications and proof techniques for serializers. 1979.

[HA86] Carl Hewitt and Gul Agha. Concurrent programming using actors: Exploiting large-scale parallelism. 1986.

[HA92] Christopher R. Houck and Gul Agha. HAL: A high-level actor language and its distributed implementation. In *ICPP (2)*, pages 158–165, 1992.

[Hal06] Philipp Haller. An object-oriented programming model for event-based actors. Master's thesis, 2006.

[HB77] Carl Hewitt and Henry Baker. Actors and continuous functionals. 1977.

[Hew71] Carl Hewitt. *Description and Theoretical Analysis (Using Schemata) of* PLANNER*: A Language for Proving Theorems and Manipulating Model in a Robot.* PhD thesis, 1971.

[Hew76] Carl Hewitt. Viewing control structures as patterns of passing messages. 1976.

[HO07] Philipp Haller and Martin Odersky. Actors that Unify Threads and Events. Technical report, 2007.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[HRAA84] Carl Hewitt, Tom Reinhardt, Gul Agha, and Giuseppe Attardi. Linguistic support of receptionists for shared resources. 1984.

[HS75] Carl Hewitt and Brian Smith. A PLASMA primer. 1975.

[Lie81a] Henry Lieberman. A preview of act 1. 1981.

[Lie81b] Henry Lieberman. Thinking about lots of things at once without getting confused. 1981.

[Lov05] Robert Love. *Linux kernel development.* Novell Press, Indianapolis, IN, USA, second edition, 2005.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[Man88] Carl Manning. Acore: The design of an actor core language and its compiler. 1988.

[Mey00] Bertrand Meyer. Principles of language design and evolution, May 12 2000.

[Mil75] Robin Milner. Processes, a mathematical model of computing agents. In *Logic Colloquium, Bristol 1973*, pages 157–174. North-Holland, 1975.

[MK99] Jeff Magee and Jeffrey Kramer. *Concurrency: State Models & Java Programs.* Wiley, 1999.

[Pai98] Jocelyn Paine. Everything is not an object, 1998.

[Que96] Christian Queinnec. *Lisp in Small Pieces.* Cambridge University Press, 1996.

[Rep92] John H. Reppy. *Higher-Order Concurrency.* PhD thesis, Cornell University, 1992.

[RH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques and Models of Computer Programming.* The MIT Press, 2004.

[Roy06] Peter Van Roy. Convergence in language design: A case of lightning striking four times in the same place. 2006.

[Seb03] Robert W. Sebesta. *Concepts of Programming Languages*. Benjamin/Cummings, sixth edition, 2003.

[SL05] Herb Sutter and James R. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[The83] Daniel G. Theriault. Issues in the design and implementation of act2. 1983.

[VA01] Carlos A. Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Notices*, 36(12):20–34, 2001.

Appendix A: Merlin Grammar

Here is the complete grammar of Merlin in extended BNF notation.

```
⟨program⟩                 ::=  ⟨statement⟩
⟨statement⟩               ::=  ⟨name_binding⟩
                           |   ⟨tell⟩
                           |   ⟨become_statement⟩
                           |   ⟨conditional⟩
                           |   ⟨statement⟩ ; ⟨statement⟩
⟨expression⟩              ::=  "("⟨expression⟩")"
                           |   ⟨expression⟩ ⟨binary_op⟩ ⟨expression⟩
                           |   ⟨behaviour_expression⟩
                           |   ⟨spawn_expression⟩
                           |   ⟨unary_op⟩ ⟨expression⟩
                           |   ⟨expression⟩ ⟨slice⟩
                           |   ⟨tuple⟩
                           |   ⟨self_reference⟩
                           |   ⟨name⟩
                           |   ⟨literal⟩
⟨self_reference⟩          ::=  self
⟨unary_op⟩                ::=  not | number? | boolean? | string? | tuple? | behaviour? | address?
⟨binary_op⟩               ::=  + | − | ∗ | / | and | or | is | < | > | = | >= | <= | ! =
⟨become_statement⟩        ::=  become ⟨expression⟩ ⟨tuple⟩
⟨conditional⟩             ::=  if ⟨expression⟩ "[" ⟨statement⟩"]" [ else "["⟨statement⟩"]" ]
⟨tuple⟩                   ::=  () | (⟨expression⟩[, ⟨expression⟩]∗)
⟨slice⟩                   ::=  {⟨expression⟩ .. ⟨expression⟩}
⟨concatenation⟩           ::=  ⟨expression⟩ . ⟨expression⟩
⟨tell⟩                    ::=  tell ⟨expression⟩ ⟨expression⟩
⟨name_binding⟩            ::=  ⟨name⟩ : ⟨expression⟩
⟨spawn_expression⟩        ::=  spawn ⟨expression⟩ ⟨tuple⟩
⟨behaviour_expression⟩    ::=  "[" [⟨formal_parameters⟩] ⟨receptor_list⟩"]"
⟨formal_parameters⟩       ::=  () | (⟨name⟩[, ⟨name⟩]∗)
⟨receptor_list⟩           ::=  ⟨receptor⟩∗
⟨receptor⟩                ::=  "["⟨pattern⟩ "|" ⟨reaction⟩"]"
⟨pattern⟩                 ::=  ⟨pattern_element_list⟩[, ⟨expression⟩]
⟨reaction⟩                ::=  ⟨statement⟩
⟨pattern_element_list⟩    ::=  () | (⟨pattern_element⟩[, ⟨pattern_element⟩]∗)
⟨pattern_element⟩         ::=  ⟨name⟩ | ⟨literal⟩
⟨literal⟩                 ::=  String | Number | Boolean
```

Semicolons between statements are optional when only one statement occupies a line. Note that this grammar is ambiguous. Disambiguation rules are given on the next page.

Since several expression types can have multiple parse trees. The grammar is disambiguated using an operator precedence hierarchy, which is given here in a highest-to-lowest precedence order:

```
* /
+ -
number?  boolean?  string?  tuple?  behaviour?  address?
> < = >= <= !=
is
or
and
not
```

Appendix B: Interpreter Source

This appendix gives the full source of the Merlin interpreter described in chapter 4. The interpreter is written for PLT Scheme, and will not work on other Scheme implementations.

```
; Merlin interpreter
; Author: Simon Kongshoj
;
; Written for PLT-Scheme. Most of this code should be fairly straightforward to
; translate into another R5RS-compliant Scheme. The threads, hash tables and
; asynchronous communication are PLT-specific.
;
; NOTE: This implementation is intended purely for demonstration purposes. It
; is deliberately kept as brief and simple as possible, in order to make it
; easier for the reader to understand the semantics of the Merlin language. It
; is not intended as an industrial tool, and leaves out many safety features
; and performance optimizations.

(require (lib "async-channel.ss"))
(require (lib "list.ss"   "srfi" "1"))
(require (lib "string.ss" "srfi" "13"))

; Actor Directory ==========================
(define max-address 0)
(define address-sema (make-semaphore 1))

(define (make-unique-address)
  (semaphore-wait address-sema)
  (set! max-address (+ max-address 1))
  (semaphore-post address-sema)
  max-address)

(define directory (make-hash-table))
(define (directory-lookup address) (hash-table-get directory address))
(define (actor-port addr) (car (directory-lookup addr)))
(define (directory-register address port control)
  (hash-table-put! directory address (list port control)))

; Core Actor Loop ==========================
(define (actor-loop addr chan script locals buffer)
  (let* ((msg (async-channel-get chan))
         (receptor (select-receptor msg script locals))
         (state (cons addr #f))
         (result (if receptor (script-exec (recept-reaction receptor)
                                           (recept-env receptor)
                                           state)))
         (next (next-behaviour state locals))
         (new-script (if next (next-script next) script))
         (new-locals (if next (next-env next) locals))
         (new-buffer (cond ((and receptor next) (retry-buffer chan buffer) '())
                           (receptor buffer)
                           (else (cons msg buffer)))))
    (actor-loop addr chan new-script new-locals new-buffer)))

(define (next-behaviour state locals)
  (if (cdr state)
    (let* ((specified-env (behaviour-env (state-behaviour state)))
           (next-env (calculate-next-environment state locals))
           (next-script (behaviour-script (state-behaviour state))))
      (cons next-script next-env))
    #f))

(define (calculate-next-environment state locals)
  (let* ((specified (behaviour-env (state-behaviour state)))
         (next-env (if (eq? specified 'this-environment) locals specified)))
    (extend-environment
      (behaviour-params (state-behaviour state))
      (state-actual-params state)
      (enclosing-environment next-env))))

(define (state-behaviour statevar) (caddr statevar))
(define (state-actual-params statevar) (cadr statevar))

(define (next-script nextbhv) (car nextbhv))
(define (next-env nextbhv) (cdr nextbhv))

(define (retry-buffer chan buf)
  (for-each (lambda (msg) (async-channel-put chan msg)) buf))

; Pattern-Matched Message Reception ========
(define (recept-reaction receptor) (car receptor))
(define (recept-env receptor) (cdr receptor))

(define (script-pattern script) (caar script))
(define (script-reaction script) (cadar script))
```

```
(define (has-guard? pattern)
  (and (pair? (last pattern)) (eq? (car (last pattern)) 'where)))

(define (guard pattern) (cadr (last pattern)))

(define (strip-guard pattern)
  (if (has-guard? pattern)
    (take pattern (- (length pattern) 1))
    pattern))

(define (check-guard pattern env)
  (if (has-guard? pattern)
    (guard-eval (guard pattern) env)
    #t))

(define (select-receptor msg script env)
  (if (null? script) #f
    (let* ((matching (match (strip-guard (script-pattern script)) msg env))
           (guard-accept? (check-guard (script-pattern script) matching))
           (match? (and guard-accept? (not (eq? matching 'no-match)))))
      (if match?
        (cons (script-reaction script) matching)
        (select-receptor msg (cdr script) env)))))

(define (match pattern exp env)
  (cond ((eq? env 'no-match) 'no-match)
        ((equal? pattern exp) env)
        ((symbol? pattern)
         (if (name-lookup pattern env)
           (if (equal? (name-exp pattern env) exp) env 'no-match)
           (extend-environment (list pattern) (list exp) env)))
        ((and (pair? pattern) (pair? exp))
         (match (cdr pattern) (cdr exp)
                (match (car pattern) (car exp) env)))
        (else 'no-match)))

; Actor Creation ==========================
(define (initial-env bhv actuals)
  (extend-environment (behaviour-params bhv) actuals (behaviour-env bhv)))

(define (actor-thread addr port bhv actuals)
  (thread (lambda ()
            (actor-loop
              addr port (behaviour-script bhv) (initial-env bhv actuals) '()))))
```

```
(define (new-actor behaviour actuals)
  (let* ((addr (make-unique-address))
         (port (make-async-channel))
         (control (actor-thread addr port behaviour actuals)))
    (directory-register addr port control)
    addr))

; Built-In Actors ========================
(define (register-primitive-actor func)
  (let* ((addr (make-unique-address))
         (chan (make-async-channel))
         (control (thread (lambda () (func chan)))))
    (directory-register addr chan control)
    addr))

(define (console-proc chan)
  (let* ((msg (async-channel-get chan)))
    (write msg) (newline)
    (console-proc chan)))

(define (void-proc chan)
  (let* ((msg (async-channel-get chan))) (void-proc chan)))

(define console-actor (register-primitive-actor console-proc))
(define void-actor (register-primitive-actor void-proc))

; Script Processor ========================
(define (script-exec exp env state)
  (cond ((self-evaluating? exp) exp)
        ((and-exp? exp) (eval-and (vals (operands exp) env state) env))
        ((or-exp? exp) (eval-or (vals (operands exp) env state) env))
        ((self-reference? exp) (cons 'address (car state)))
        ((name? exp) (name-exp exp env))
        ((definition? exp) (eval-definition exp env state))
        ((if? exp) (eval-if exp env state))
        ((tell? exp) (eval-tell exp env state))
        ((spawn? exp) (eval-spawn exp env state))
        ((become? exp) (eval-become exp env state))
        ((behaviour-definition? exp)
         (make-behaviour (behaviour-parameters exp) (behaviour-body exp) env))
        ((begin? exp) (eval-sequence (begin-actions exp) env state))
        ((operation? exp)
         (apply-operation (script-exec (operator exp) env state)
                          (vals (operands exp) env state)))
        (else (error "ERROR in script-exec! Unknown expression: " exp))))
```

```
(define (guard-eval exp env)
  (let ((state '()))
    (cond ((self-evaluating? exp) exp)
      ((and-exp? exp) (eval-and (vals (operands exp) env state) env))
      ((or-exp? exp) (eval-or (vals (operands exp) env state) env))
      ((name? exp) (name-exp exp env))
      ((operation? exp)
       (apply-operation (script-exec (operator exp) env state)
                     (vals (operands exp) env state)))
      (else (error "ERROR in guard-eval! Unknown expression: " exp)))))

(define (apply-operation rator rands) (apply (car rator) rands))

(define (vals exps env state)
  (if (no-operands? exps)
    '()
    (cons (script-exec (first-rand exps) env state)
          (vals (rest-rands exps) env state))))

; Statements ==============================
(define (eval-tell exp env state)
  (let* ((values (vals (operands exp) env state))
         (addr (cdar values))
         (msg (cadr values)))
    (async-channel-put (actor-port addr) msg)))

(define (eval-sequence exps env state)
  (cond ((last-exp? exps) (script-exec (first-exp exps) env state))
        (else (script-exec (first-exp exps) env state)
              (eval-sequence (rest-exps exps) env state))))

(define (eval-if exp env)
  (if (script-exec (if-predicate exp) env state)
    (script-exec (if-consequent exp) env state)
    (script-exec (if-alternative exp) env state)))

(define (eval-spawn exp env state)
  (let* ((values (vals (operands exp) env state))
         (bhv (car values))
         (actuals (cadr values)))
    (cons 'address (new-actor bhv actuals))))

(define (eval-become exp env state)
  (set-cdr! state (list (script-exec (become-params exp) env state)
                        (script-exec (become-script exp) env state))))
```

```
(define (eval-definition exp env state)
  (if (and
        (pair? (definition-value exp))
        (eq? (car (definition-value exp)) 'behaviour))
    (begin
      (let ((new-env (deep-copy env)))
        (define-name! (definition-name exp)
                          (list 'behaviour
                                (behaviour-parameters (definition-value exp))
                                (behaviour-body (definition-value exp))
                                'this-environment)
                          new-env)
        (define-name! (definition-name exp)
                          (list 'behaviour
                                (behaviour-parameters (definition-value exp))
                                (behaviour-body (definition-value exp))
                                new-env)
                          env)))
    (begin
      (define-name! (definition-name exp)
                        (script-exec (definition-value exp) env state)
                        env))))

; Expressions ==============================
(define (name-exp var env)
  (let ((val (name-lookup var env)))
    (if val val (error "Unbound name" var))))

; Abstract Syntax ==========================
(define (tagged-list? exp tag) (if (pair? exp) (eq? (car exp) tag) #f))

(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
(define (first-rand ops) (car ops))
(define (rest-rands ops) (cdr ops))

(define (self-reference? exp) (eq? exp 'self))

(define (tell? exp) (tagged-list? exp 'tell))
(define (spawn? exp) (tagged-list? exp 'spawn))

(define (definition? exp) (tagged-list? exp 'define))
(define (definition-name exp) (cadr exp))
(define (definition-value exp) (caddr exp))

(define (become? exp) (tagged-list? exp 'become))
```

```
(define (become-script exp) (cadr exp))
(define (become-params exp) (caddr exp))

(define (behaviour-definition? exp) (tagged-list? exp 'behaviour))
(define (behaviour-parameters exp) (cadr exp))
(define (behaviour-body exp) (cddr exp))

(define (begin? exp) (tagged-list? exp 'begin))
(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))

(define (if? exp) (tagged-list? exp 'if))
(define (if-predicate exp) (cadr exp))
(define (if-consequent exp) (caddr exp))
(define (if-alternative exp) (if (not (null? (cdddr exp))) (cadddr exp) #f))

(define (or-exp? exp) (tagged-list? exp 'or))
(define (and-exp? exp) (tagged-list? exp 'and))

(define (name? exp) (symbol? exp))

(define (operation? exp) (pair? exp))
(define (no-operands? ops) (null? ops))

(define (self-evaluating? exp)
  (cond ((number? exp) #t) ((string? exp) #t) (else #f)))

; Behaviours ==============================
(define (make-behaviour params script env)
  (list 'behaviour params script (deep-copy env)))

(define (behaviour-params bhv) (cadr bhv))
(define (behaviour-script bhv) (caddr bhv))
(define (behaviour-env bhv) (cadddr bhv))

; Boolean operators =======================
(define (eval-or exps env)
  (cond
    ((car exps) #t)
    ((null? (cdr exps)) #f)
    (else (eval-or (cdr exps) env))))

(define (eval-and exps env)
  (cond
    ((not (car exps)) #f)
    ((null? (cdr exps)) #t)
    (else (eval-and (cdr exps) env))))

; Type Predicates ==========================
(define (tuple? datum)
  (and
    (pair? datum)
    (not (eq? (car datum) 'address))
    (not (eq? (car datum) 'behaviour))))

(define (behaviour? datum)
  (and
    (pair? datum)
    (eq? (car datum) 'behaviour)))

(define (address? datum)
  (and
    (pair? datum)
    (eq? (car datum) 'address)))

; Various Operations =======================
(define (merlin-length object)
  (cond
    ((string? object) (string-length object))
    ((tuple? object) (length object))))

(define (join obj1 obj2)
  (cond
    ((and (string? obj1) (string? obj2)) (string-append obj1 obj2))
    ((and (tuple? obj1) (tuple? obj2)) (append obj1 obj2))))

(define (slice obj start end)
  (cond
    ((string? obj) (substring start end))
    ((tuple? obj) (take (drop obj start) (- end start)))))


; Primitives and Operators =================
(define (primitive-actor-names) (map car primitive-actors))
(define (primitive-actor-addrs) (map cadr primitive-actors))
(define primitive-actors
  (list
    (list 'console (cons 'address console-actor))
    (list 'void (cons 'address void-actor))))
```

```
(define (operator-names) (map car operators))
(define (operator-objects) (map cdr operators))

(define operators
  (list
    (list '+ +)
    (list '- -)
    (list '* *)
    (list '/ /)
    (list '= =)
    (list '> >)
    (list '< <)
    (list '>= >=)
    (list '<= <=)
    (list 'not not)
    (list 'length merlin-length)
    (list 'slice slice)
    (list 'join join)
    (list 'tuple list)
    (list 'boolean? boolean?)
    (list 'number? number?)
    (list 'string? string?)
    (list 'tuple? tuple?)
    (list 'behaviour? behaviour?)
    (list 'address? address?)
    (list 'is equal?)))

; Environments ============================
(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define empty-environment '())

(define (make-frame vars vals) (cons vars vals))
(define (frame-names frame) (car frame))
(define (frame-values frame) (cdr frame))
(define (add-binding! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))

(define (deep-copy data)
  (if (not (pair? data))
    data
    (cons (deep-copy (car data))
          (deep-copy (cdr data)))))
```

```
(define (name-lookup name env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? name (car vars)) (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env empty-environment)
      #f
      (let ((frame (first-frame env)))
        (scan (frame-names frame) (frame-values frame)))))
  (env-loop env))

(define (define-name! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars) (add-binding! var val frame))
            ((eq? var (car vars)) (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-names frame) (frame-values frame)))
  env)

(define (extend-environment vars vals base)
  (if (= (length vars) (length vals))
    (cons (make-frame vars vals) base)))

(define (make-environment)
  (extend-environment (append (primitive-actor-names)
                              (operator-names))
                      (append (primitive-actor-addrs)
                              (operator-objects))
                      empty-environment))

(define universe-environment (make-environment))

(define (merlin-exec exp) (script-exec exp universe-environment #f))
```