Using Abstraction in Explicitly Parallel Programs

by

Katherine Anne Yelick

This report is a revised version of the author's thesis, which was submitted to the Department of Electrical Engineering and Computer Science on December 31, 1990 in partial fulfillment of the requirements for the degree of Doctor of Philosophy at the Massachusetts Institute of Technology. The thesis was supervised by John V. Guttag. The author's current address is the Computer Science Division, University of California, Berkeley, CA 94720.

Abstract

It is well-known that writing parallel programs that are both fast and correct is significantly harder than writing sequential ones. In this thesis we introduce a *transition-based* approach to the design and implementation of parallel programs. This approach is aimed at applications whose complex data and control structures make them hard to parallelize by conventional means. It is based on a programming model with explicit parallelism, and it incorporates data and process parallelism within a uniform framework.

The transition-based approach addresses the problem of program synthesis by breaking the development process into four distinct phases, each with explicit correctness and performance requirements. Module interfaces are well-defined so that rigorous correctness arguments can be made when desired. Application-specific scheduling is used to enhance performance, and significant performance tuning of the scheduler can be done in the last phase of development.

Programs designed with this approach rely on data abstractions whose operations behave serially, but have highly concurrent implementations. Specifications and implementations of several such abstractions are presented. Some of the implementations are notable in that they allow access to shared variables without explicit synchronization, thereby exploiting the full power of sequentially consistent shared memory. To define correct behavior, we consider correctness notions in the literature and present two new notions that address performance concerns. First, we liberalize the notion of safety property of linearizability by incorporating *interference specifications*, which make assertions about operations should not be run concurrently. Second, we define a relatively weak liveness property, *non-stopping*, that makes efficient implementations possible.

We apply our programming method to two example programs, matching of terms and completion of rewriting systems. Both programs are designed and performance tuned using the transition-based approach. Their implementations make use of highly concurrent types that meet our correctness conditions, and they perform well on a small shared-memory multiprocessor. The completion program is interesting in its own right, as it is the first parallel solution to an important and much studied problem.

Keywords: Parallel Programming, Multiprocessor, Programming Methodology, Specification, Refinement, Concurrent Object, Linearizability, Sequential Consistency, Scheduling, Completion of Term Rewriting Systems, Term Matching, Interference Specifications, Transition-Based Development

Contents

Acknowledgements

I doubt that the work presented here would have been started without the guidance of my advisor, John Guttag, and I am quite sure that his support was essential to seeing the thesis completed. Over the years John has acted as supervisor, mentor, career counselor and friend, and for each of these I am deeply grateful—he always seemed to know the right words of advice or encouragement. I would also like to thank him for promptly reading drafts of the chapters, even when it meant working evenings, weekends, or holidays.

Each of my committee members played an invaluable role in producing this thesis. Bill Weihl showed an interest in my work from its early stages on, and took the time to have detailed technical discussions that were both informative and inspirational. Barbara Liskov was very encouraging, and offered keen insight into the methodological issues raised by my approach. Nancy Lynch displayed her usual attention to detail in reading a draft of this thesis. I appreciate her help in clarifying a number of definitions and in determining which formal notions were essential to the thesis.

I would like to thank all of the members of John's research group, particularly the current members (Steve Garland, Daniel Jackson, Mark Reinhold, and Mark Vandevoorde) for their feedback on my ideas about parallel programming. Mark Vandevoorde deserves special thanks for answering my Firefly-specific questions, for maintaining our Firefly environment, and for providing moral support when I was forced to do hardware repair. I would also like to thank Steve Garland and Ursula Martin for reading Chapter 5, and for contributing some of the more interesting examples for parallel completion. Thanks are also due to the Digital Equipment Corporation's Systems Research Center, which provided the parallel computing environment.

Many of my friends provided a welcome distraction from work. The members of the graduate rowing team were a constant source of inspiration and support; I want to thank coach Stu Schmill for his ingenuity in designing our workouts, and Martha Gray and Jennie Kwo for being crazy enough to do them. Paul Grimshaw, Jim O'Toole, Mark Vandevoorde, and Jennifer Welch all lent sympathetic ears and gave helpful advice at various times.

Finally, I would like to thank everyone in my family for encouragement in whatever I pursued. They offered a much needed perspective, were examples of strength and good humor, and displayed admirable patience in waiting for me to finish.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-89-J-1988, by the National Science Foundation under grant CCR-8910848, and by the Digital Equipment Corporation.

Chapter 1

Introduction

Writing parallel programs that are both correct and fast is significantly harder than writing sequential ones. In this thesis we study the specifics of what makes parallel programs difficult to build, and we propose an approach that simplifies their construction.

The approach is designed for building symbolic applications, and it makes use of *explicit*, rather than *implicit*, parallelism. With explicit parallelism, decisions about what to parallelize and when to synchronize are made by the programmer; implicit parallelism leaves these decisions to the compiler and run-time system. While implicit parallelism is easier to use, there is little evidence to suggest that it can produce fast implementations for the kinds of applications our work addresses. Our approach also assumes an imperative, rather than functional, programming style. Functional programs avoid many of the correctness problems that come with parallelism, but we believe that the use of shared mutable data is important to achieving good performance.

We apply our approach to two symbolic applications: matching of terms, and completion of term rewriting systems. The programs have clean modular structures and exhibit good performance on a small multiprocessor. In presenting the applications, we describe the development process from the early design stages through reasoning about the correctness of the designs and performance tuning the implementations.

1.1 The Problem

In this section we study the task of parallel programming, first in a general context and then with respect to symbolic programs. Section 1.1.1 addresses the question of why parallel programming is hard in general. Section 1.1.2 describes symbolic applications and why they represent a special challenge for parallelism.

1.1.1 Why is Parallel Programming Hard?

Many parallel programs suffer from one of two problems: the programs are too complicated, or the programs are too slow. Unfortunately, these two problems are not mutually exclusive.

A common complaint about parallel programs is that the increased nondeterminism that comes from parallelism renders them too complicated. Nondeterminism contributes to algorithmic complexity and makes testing and debugging more difficult; it also affects basic ideas about specification and abstraction. There is a fundamental difference between sequential and parallel program abstractions, since implementations that are correct when used sequentially may behave unpredictably when used in parallel. A notion of correctness for program modules becomes more complicated when it must allow for concurrency.

In addition to problems with the correctness of parallel programs, there are problems with their performance. An application running on a parallel machine rarely achieves the kind of performance quoted for that machine. The most obvious reason for disappointing performance is that the program lacks sufficient parallelism, either because the problem is inherently sequential or because the chosen implementation does not uncover the underlying parallelism. Programmers are often surprised by insufficient parallelism when they set out to implement a program containing highly parallel algorithms; the parallel algorithms may be fast, but overall program performance is limited by some sequential task. Even a small amount of parallelism within the limiting task may be more beneficial than a large amount additional parallelism within the others.

A second reason for poor performance is exactly the opposite: the program has too much parallelism. Contrary to idealized models of parallel computation, real machines exact a non-negligible payment for creating and managing parallel tasks. When those tasks have interdependencies, there is additional overhead for the required communication. Overhead is also

increased when the amount of parallelism in the program exceeds the parallel resources of the machine, since n-fold parallelism must be simulated on fewer than n processors using context switching. As a result, the overhead of parallelism can easily outweigh its benefit.

A final class of performance problems is characterized by *fake parallelism*: many parallel tasks exist, but they do little useful work. Fake parallelism can occur when there is too much synchronization, resulting in tasks that are either blocked or spinning. A more subtle form of fake parallelism occurs when parallel tasks are computing new results but not progressing towards an answer. If there is insufficient communication between tasks, then redundant or unnecessary work may be done.

Most program development efforts require trade-offs between performance and simplicity. However, the points discussed above indicate that the problems are magnified in the case of parallel programs, so a balance is harder to find. Anecdotal evidence from programmers confirms this claim. It is not unusual to hear reports of parallel implementations that run slower than their sequential counterparts, or of efficient parallel programs in which a seemingly innocuous change results in a program that is no longer correct.

1.1.2 Why are Symbolic Applications Particularly Hard?

Parallel programs can be characterized by the kind of parallelism they exhibit: *process parallelism* occurs when different tasks are executed in parallel, while *data parallelism* occurs when identical tasks are executed in parallel on different data. Parallel programs are also distinguished by the size of the tasks that are executed in parallel, with the extremes being *fine grained* and *coarse grained*. A *task* in our terminology is a logical unit of computation that is relatively short-lived. Tasks are executed by *threads*, which are lightweight processes that can share objects and are relatively long-lived.

Symbolic applications differ in character from numerical ones, because numerical applications tend to have more regularities that can be exploited during parallelization. As a result, programming techniques that are appropriate to one class of applications are not necessarily appropriate to another. (In discussing the differences between the two problem domains we over-simplify both: some symbolic applications are regular, some numerical ones are irregular, and many applications involve a mixture of both styles.)

- Numerical applications have more regular patterns of control flow, thereby enabling data parallelism. When one sequence of operations is applied repeatedly to different data, the repetition can be changed to data parallel code. Symbolic applications require more process parallelism, and even when high-level data parallelism can be applied, the low-level instruction streams that are taken in each instance may be different. Thus, machines supporting multiple instruction streams appear better suited to symbolic programs.

- Numerical programs typically involve large regular data structures such as arrays of numbers, while symbolic programs involve irregular structures such as unbalanced trees or graphs. Data parallelism is therefore easier to schedule in numerical programs, where the elements in an array have the same size, so granularity can be adjusted uniformly by working on fixed-size sub-arrays. In addition, symbolic operations often exhibit a kind of *performance instability*, in that a small change in the input results in a large variance in performance. These factors make it difficult to accurately predict the performance of operations within symbolic programs, so that superior performance requires making decisions about granularity and scheduling "on the fly."

- A related problem with parallelizing symbolic applications is that communication patterns between parallel tasks are irregular. Thus, while numerical applications may be implemented efficiently using a synchronous programming model, in which independent tasks run concurrently and all communication is done at global synchronization points, this model is not practical for most symbolic applications. A better model for symbolic applications involves asynchronous tasks that communicate through shared mutable data structures, which means an imperative programming language and explicit synchronization.

1.2 Contributions

As discussed above, many symbolic applications are best parallelized using a programming model with explicit parallelism and imperative constructs. The main argument against this model is that it places too great a burden on the programmer. This thesis shows that the programming complexity of parallelism can be managed using abstraction, without unduly

degrading performance. We present significant sample programs that exhibit both modularity and good performance and that, more importantly, illustrate a general approach to parallel program development.

The contributions of this thesis fall into four areas: 1) understanding modularity in parallel programs, 2) synthesizing parallel programs, 3) designing algorithms for concurrent data structures, and 4) evaluating underlying system support (both hardware and software) for parallel programming.

Our first contribution is directed at a better understanding of modularity in parallel programs. A modular program is composed of program units, or abstractions, that can be separately specified, designed, implemented, and proved correct. Using abstractions does not eliminate the complexity introduced by parallelism, but it does simplify the program by breaking it into manageable pieces. We show that with parallelism, basic units of abstraction must be restricted, notions of correctness changed, and specifications augmented. In particular, we examine correctness notions that have been proposed by others [Lam80, Lam79, HW90, AH90b], compare them on practical grounds, and offer a new correctness notion of our own. Our notion is based on *linearizability* [HW90], but is strictly weaker. It incorporates *interference specifications*, which constrain the use of an abstraction in order to admit implementations that are simpler and more efficient. Interference specifications can be used to give precise interface specifications for a common kind of object that we call a *multi-ported object*, i.e., a concurrent object that can be accessed by only a fixed number of concurrent processes.

Our second and most significant contribution is the *transition-based* approach for synthesizing parallel programs, which uses both data and process parallelism, and is intended for developing coarse-grained parallel programs. The development process is broken into four distinct steps with a statement of requirements for each step. The approach makes use of concurrent data types for modularity, and it addresses performance concerns by supporting program-level parallelism, application-specific scheduling, and fine-grained synchronization. The approach encourages the programmer to "think in parallel" by requiring that the program be described by a set of nondeterministic state transitions. Two program examples illustrate our approach to program synthesis. The examples are developed starting from the initial design phase and extending through low-level coding decisions and performance tuning. The examples show that

good performance can be achieved in modular programs.

The early stages of the approach use the same refinement methods that are proposed for Unity [CM88], but our goals in refinement are different. Unity is used as a programming language, and the presumption is that Unity compilers can be developed to generate efficient parallel code from an appropriately refined Unity program, but the practicality of the approach has not been demonstrated. Our *transition axiom specifications* are used as a design language, and our presumption is that programmers can be trained to recognize good designs at the specification level, and then implement those designs efficiently in a more conventional programming language. We present criteria to help programmers recognize good designs and programming techniques to help them produce efficient implementations. One reason for using transition-axiom specifications as a design language is that standard techniques exist for reasoning about such specifications. Correctness arguments can be made at various stages in program development using abstraction functions and invariants [Lam83, LT87, CM88]. We include examples of correctness arguments to illustrate their integration into the development process.

This thesis also contributes a number of parallel algorithms, which can be divided into two different classes: algorithms that use parallelism internally to reduce latency, and algorithms that support such parallelism by providing high throughput for access to concurrent objects. Parallel algorithms occur only at the program level in the transition-based approach, so there are only two parallel algorithms presented here: matching and completion. The completion procedure is the first parallel solution to a widely used and much studied problem in term rewriting. The matching algorithm is of pedagogical interest only, since the inputs have to be quite large before parallelism becomes advantageous. In addition, this thesis contains new algorithms for implementing concurrent data structures, including unbounded *queues* and *mappings*.

The final contribution is a set of programming techniques for writing parallel programs, addressing some of the low-level engineering issues of shared memory multiprocessors. Our data type implementations frequently use shared memory without explicit synchronization, so while the overall programming approach does not depend on shared memory, some of the code does depend on the full power of a shared (sequentially consistent) memory. These implementations provide evidence for the power of the shared memory programming model. Within the architecture community, proponents of weaker memory models claim that accesses to shared

memory in are usually protected by critical regions. Our examples contradict this claim.

1.3 Overview

In this section we give an overview of each of the chapters in the thesis, and explain the relationship between them. A comment on the treatment of related work is also appropriate at this point: since each of the main chapters touches on different fields within the broad area of parallel programming research, related work is discussed throughout the thesis at the place it is most relevant.

1.3.1 Specifying Concurrent Data Types

In Chapter 2 we consider some of the general questions related to writing modular parallel programs. What kinds of modules are useful when writing parallel programs? What do their specifications look like? What does it mean for an implementation to satisfy an interface specification? We show how many of the basic concepts such as specifications and correctness do not carry over directly from sequential programming methods, and we give extensions to the parallel domain.

A key to making abstraction work is choosing the right notion of correctness. Many correctness notions have been defined for systems involving concurrency. We consider sequential consistency [Lam79] and linearizability [HW90], and compare them from the perspective of both user and implementor. Our conclusion is that sequential consistency is adequate for defining the interface of shared memory, but that linearizability, which is strictly stronger, is needed for program modules.

We also show that good performance sometimes requires data abstractions that do not meet either of the correctness notions, because in actual implementations one is guaranteed that their use involves limited concurrency. In Chapter 2 we suggest ways of augmenting specifications with *interference specification* so that these abstractions are considered correct if they are used in a controlled environment without interfering concurrency. We also present a notions that is analogous to the implicit assumption in sequential programs that correct procedures must terminate. In parallel programs a procedure may communicate with its environment, so

assumptions about the behavior of the environment are needed. Chapter 3 contains examples of these abstractions to demonstrate that the correctness notions defined in Chapter 2 allow for simple and efficient implementations.

1.3.2 Implementing Concurrent Data Types

In Chapter 3 we describe implementations for several concurrent data objects, and identify programming techniques that can be used in general. The types are shown to be useful, since each is used in Chapter 4 or 5, and they are also reusable, since each is given with a interface specification including any interference constraints. For some of the more interesting implementations, correctness arguments are sketched. The object types include *locations*, *accumulators*, *counters*, *locks*, (unbounded) *queues*, and *mappings*. For some of these types, implementations have already been described in the literature; in those cases we include a comparison between the previously published implementations and ours.

In addition to being interesting in their own right, these implementations also illustrate some general principles of parallel programming. The most widely accepted method for implementing concurrent data types is the use of critical regions. For parallel programs in which most computation involves shared objects, placing all accesses in critical regions results in fake parallelism, since threads must wait for access to any shared object. [Her90] gives a different technique for implementing concurrent objects; it allows both mutating and non-mutating operations to proceed in parallel, but mutating operations may have to redo their work if another mutating operation is executing concurrently. Both techniques as useful in certain situations, but neither should be used exclusively. We offer other techniques, and describe the performance trade-offs that make one technique better than others in a given circumstance.

One of the implementations, that of the *mapping* type, is particularly interesting. First, the implementation only allows for access by a fixed number of concurrent threads, which gives its specification a unique kind of interference constraint. We call these objects *multi-ported objects*. In our programs, multi-ported objects are used to reduce contention; we believe such objects are also useful for achieving locality of access on architectures in which memory is not a uniform distance from each processor. The implementation of the *assign* operation on *mappings* is also novel, as it uses backoff to handle collisions in key assignments; it was chosen after more

obvious implementations had been tried and discarded.

1.3.3 Designing Parallel Programs

In Chapter 4 we present *transition-based development* of parallel programs. Fundamental to the approach is the separation of correctness and performance concerns; the approach has four distinct steps, each having explicit correctness requirements and performance goals. Modularity is achieved by requiring that data type implementations meet the correctness conditions presented in Chapter 2. Good performance is achieved by addressing each of the performance concerns outlined in Section 1.1.1:

- *Insufficient parallelism.* In the transition-based approach, data and process parallelism are supported within a single framework. In the earliest stages of design, the programmer is encouraged think about a parallel solution to the problem, so that synchronization and serialization are added only as necessary. This technique uncovers *program level parallelism*, i.e., parallelism within the highest level algorithm. In contrast, methods that start from a traditional sequential programming model tend to retain a sequential algorithm at the program level, and they may result in unnecessary synchronization points that are artifacts of the development process.

- *Over-abundant parallelism.* Part of the overhead from thread creation and management comes from using an overly general system scheduler. By using *application-specific scheduling*, which runs on top of a fixed set of system-provided threads, this overhead can be avoided. Favoring coarse over fine grained parallelism also avoids some of the cost of parallelism. By allowing for performance tuning (e.g., scheduling and granularity adjustments) late in the program development process, a balance can be found between too much parallelism and not enough parallelism.

- *Fake parallelism.* The underlying causes of fake parallelism are lack of communication between tasks, which results in unnecessary work, and too much communication, which results in thread idle time while waiting for communication. Our strategy is to have frequent communication through shared mutable objects, but to control access using short critical regions (usually a constant number of instructions). Procedures are never required

to block (or wait) as part of their specification; instead, procedures return quickly with an exception, indicating to the caller that the operation could not be performed.

In the transition-based approach, a program is described at a high level by a transition-axiom specification, which is comprised of a set of transition axioms that define the legal state changes of the program. The first transition-axiom specification should reflect the programmer's intuition of what basic tasks must be performed, but it may not describe an efficient program, or it may not be implementable by the rules of our approach. The second step of development is to refine the transition axiom specification into another transition axiom specification that corresponds to an efficient implementation. In Chapter 4 we give criteria for such specifications and describe specific refinements that help meet those criteria. In the third step of development, the programmer implements a *transition procedure* for each transition axiom; the procedure performs a state change that is consistent with the transition axiom. The procedures must be synchronized to ensure that any concurrent execution is correct, in the sense that no illegal state may be reached. The last step of program development is to determine the task grain size and describe a dynamic scheduling strategy for the transition procedures. In practice, the process of choosing a level of granularity and a scheduling strategy is iterative, and usually involves a number of rounds of analysis and tuning.

The transition-based approach is applied to the term matching problem, with thorough discussion of how the program is designed, implemented, and performance tuned. We present careful correctness arguments for pieces of the design, demonstrating a modular approach to reasoning about this style of program. Performance numbers are presented based on experiments run on a five processor Firefly [TSJ87].

1.3.4 Parallel Completion

Chapter 5 contains a proof of concept for the transition-based approach. We give a parallel solution to the completion problem for term rewriting systems, bringing together the work in the earlier chapters. The design and implementation is done using the transition-based approach outlined in Chapter 4, notions of abstraction and correctness defined in Chapter 2, and concurrent object specifications and implementations given in Chapter 3. The program produced is reasonably large (about ten thousand lines of C), modular, and efficient.

The completion problem displays interesting properties for parallelism that are typical of many symbolic programs, so the existence of a carefully designed parallel solution should offer information on the use of parallelism across a spectrum of symbolic applications. The Knuth-Bendix procedure, a sequential solution to the completion problem, has been extensively studied, modified, and extended. (See [Buc85] for a historical survey of completion procedures, with more than 200 references that include algorithms, applications, and implementations.) Our procedure is significant from an algorithmic standpoint, since the parallel procedure differs in non-trivial ways from sequential ones.

The implementation of our design demonstrates that it is of practical, as well as theoretical, interest. The implementation was run on a Firefly with six CVAX processors. On all realistic examples, including several taken from the literature, parallelism proves to be beneficial. On some larger examples the performance gain from six-fold parallelism is between four and five.

Chapter 2

Specifying Concurrent Data Types

The programming approach introduced in Chapter 4 depends on having data type implementations that can be treated abstractly by the rest of the program. In general, abstraction is used to hide unnecessary implementation details from a user; in parallel programs, low level concurrency is often one of those details. A *concurrent data type* has operations that can be viewed as executing indivisibly, when in fact they may execute concurrently with one another. Note that the parallelism is in the *use* of a concurrent data type; whether or not individual operations are implemented using parallelism is orthogonal to the question of whether operations can be invoked concurrently.

In this chapter we consider the problem of specifying concurrent data types and address a number of questions. What do specifications of concurrent data types look like? What does it mean for an implementation to be correct with respect to a specification? How does the choice of a correctness notion restrict implementations? These questions have already been asked and answered for sequential data types; we are interested in how the answers differ for parallel programs.

There are a number of concepts in this chapter that are essential to the rest of the thesis. The first is a notion of correctness for concurrent data types, *linearizability modulo an interference specification*, which generalizes the notion of *linearizability* defined by Herlihy and Wing [HW90]. Linearizability places a requirement on all operations of a data type, implicitly assuming that all operations may be invoked concurrently. Our generalization allows operations to *interfere*, meaning their behavior is undefined if they are invoked concurrently; we incorporate

this interference information into the specification of a data type to facilitate code reuse. The result is a correctness condition that is potentially as powerful as linearizability, but can be tailored to the needs of a particular kind of concurrency. In Chapter 3, we give examples of data types with interference relations, and demonstrate that they can sometimes allow for a simpler or more efficient implementation.

The second important concept in this chapter is another correctness notion called *non-stopping*; it is analogous to the termination requirement on sequential operations but allows for nonterminating executions in certain pathological cases that can be avoided in practice. Non-stopping is strictly weaker than the related notion of *wait-freeness* defined by Herlihy [Her88]. A key difference between the two notions is that wait-freeness requires termination in the presence of processor failures, whereas non-stopping places no constraints on termination in the presence of failures. Wait-free data types are attractive in principle, but they often require complicated algorithms and bear a significant performance overhead.[1] Non-stopping data types have lower overhead, and the notion corresponds to the unstated condition used by many programmers in practice: non-stopping operations cannot dead-lock, and any finite number of invocations must eventually terminate.

In addition to these key notions, this chapter contains material that supports the rest of the thesis. In Section 2.1, we describe an approach to writing specifications for concurrent data types that will be used in Chapters 3, 4, and 5. Specifications are used in this chapter to demonstrate the difference between various correctness notions and in the other chapters to describe the modules in our programs.

The final piece of background material in this chapter is a formal model for concurrent data types, which is an adaptation of the model used by Herlihy and Wing [HW90]. The model is presented in Section 2.2, along with an intuitive pictorial version that is used for the examples. We have tried to include sufficiently detailed informal descriptions and enough examples that the reader may skip the formal definitions and still read the rest of the thesis.

The correctness notions are defined in Sections 2.3 through 2.5. Section 2.3 gives two definitions from the literature: sequential consistency [Lam79] and linearizability [HW90]. We compare the two notions and explain our choice of linearizability as the basis of correctness

[1] Herlihy and Tuttle [HT90] show theoretical lower bounds on the performance of wait-free data types.

for concurrent data types in this thesis. In Section 2.4, we argue that the original definition of linearizability is too restrictive, and therefore add the ability to qualify linearizability by adding an interference specification. Section 2.5 addresses the problem of termination (or more generally *liveness*) for concurrent data type operations, and presents our notion of non-stopping implementations. We end the chapter in Section 2.6 with a discussion of the key points and some related work.

2.1 An Approach to Writing Specifications

An *abstraction* is a unit of program text that can be independently designed, implemented, tested, and proved correct. Each of these activities is done with respect to an *interface specification* that constrains the behavior of the program text. Thus, the ability to perform abstraction is intimately tied to the expressiveness of interface specifications.[2]

One approach to writing interface specifications is to use conventional specifications that describe serial data types, but to change the notion of correctness to allow for concurrency among operations. We employ this approach here. It is similar to Lamport's approach to specifying concurrent program modules [Lam83, Lam89], except that his specifications include a type-specific liveness constraint, while we use a fixed liveness property on all types.

A correctness notions for concurrent objects depends on a correctness notion for serial objects with respect to some serial specifications; the choice of specification language and serial correctness notion are largely independent of concurrency considerations. Our specifications are informal although they have a similar structure to interface specifications written in the Larch family of formal specification languages [GHW85, Win83]. Others have used Larch-style specifications to specify concurrent operations [HW90, BGHL87]. We describe the meaning of our specifications and a serial correctness notion through an example.

Figure 2-1 shows an example of an interface specification for a *container* type. (We present our programs in a variant of the CLU programming language [LAB+81], discussed in further detail in Chapter 3.) The *container* specification is polymorphic; the *type* parameter is used to

[2]In Chapters 4 and 5 we use another kind of specification, transition axiom specifications, that specify parallel algorithms. Here and in Chapter 3, where all specifications define data type interfaces rather than algorithms, we sometimes refer to interface specifications more simply as *specifications*.

container = **datatype** [t: **type**] **has** create, insert, choose

create = **procedure** () **returns** (container)
ensures: returns an empty container

insert = **procedure** (c: container, e: t) **returns** (int)
ensures: adds e to c

choose = **procedure** (c: container) **returns** (t) **signals** (empty)
ensures: if c is empty then the empty signal is raised,
otherwise an element of c is removed and returned

waiting_choose = **procedure** (c: container) **returns** (t)
when: c is not empty
ensures: an element of c is removed and returned

conditional_choose = **procedure** (c: container) **returns** (t)
requires: c is not empty
ensures: an element of c is removed and returned

Figure 2-1: Specification of a container type.

allow containers of any type of element. The *insert* operation takes an element of type t, and the three different *choose* operations return an element of type t. The specifications of *choose*, *waiting_choose*, and *conditional_choose* describe operations that have the same behavior on non-empty containers: an element of the container is chosen, deleted, and returned. However, the three operations differ in their behavior on empty containers. The *choose* operation will raise the signal *empty* if the container is empty. Signals in CLU are considered alternate forms of termination for an operation, and need not cause the program to halt if the calling procedure has code to handle the raised signal; an operation cannot, however, be resumed once it has signaled. The other two forms of *choose* are described below.

Each operation on a data type is specified by three components: an *ensures* clause, a *requires* clause, and a *when* clause. The *requires* and *when* clauses are optional; a missing clause has the same meaning as the clause with the single predicate *true*. Together the three clauses define a relation on pairs of states, called the *pre* and *post* states. The clauses are used as follows:

1. A *requires* clause places a constraint on the (abstract) input values in the state before the operation executes (the *pre* state). It is the caller's responsibility to make sure the *requires* clause is true, and if it is not true, the behavior of the operation is unconstrained.

The specification of *conditional_choose* contains a *requires* clause with the condition that c is non-empty. Thus, the implementor of *conditional_choose* may assume that c is never empty. In parallel programs, it is rarely useful to have a *requires* clause on a mutable shared object, because the value of such an object can change between the time the user invokes the operation and the time the operation begins executing. It is therefore difficult for the caller to guarantee that *requires* condition will be true when the operation starts executing.

2. A *when* clause also places a constraint on the (abstract) input values in the *pre* state. However, the responsibility for satisfying a *when* clause is with the implementation of the operation, rather than the caller. The implementation is not allowed to take any observable action until the *when* clause is true. In sequential programs, *when* clauses are not useful, because if an operation is invoked with a false *when* clause, it will never terminate–it must "hang." In parallel programs, *when* clauses can be quite useful; for example, the *when* clause in the *waiting_choose* specification states that c must be non-empty, if the operation is invoked with c empty, it must wait for some other thread to add elements before mutating c or returning. We refer to specifications with non-trivial *when* clauses as *waiting specifications*.

3. The *ensures* clauses relates the *pre* state to the set of possible *post* states, which exist when the procedure returns. In Figure 2-1, all of the *choose* specifications are nondeterministic in the choice of element to be deleted, so there are many possible *post* states for a given *pre* state. The responsibility for the *ensures* clauses resides with the implementor of the operation.

A *state* is an assignment of values to objects, and an *execution* is a possibly infinite sequence of states, with each consecutive pair of states representing the *pre* and *post* states of some procedure invocation. A more precise characterization of states for a real programming language, and interpretations of predicates in those states, would require a formal language and a richer model than we wish to pursue here. Wing [Win83] gives such an interpretation for formal specifications in sequential CLU programs, and Goldman [GL90] gives a detailed model of shared state for concurrent operations.

Given a specification for some procedure P, a state pair $\langle pre, post \rangle$ is said to be a *legal* execution of P if either:

1. the *requires* clause is false in *pre*, or

2. the *when* clause is true in *pre* and the *ensures* clause is true in the pair *pre* and *post*.

Note the difference between the *requires* clause and the *when* clause: if the *requires* clause is false in *pre*, then any *post* state is legal, but if the *when* clause in false in *pre*, then no *post* state is legal. A data type specification is given by a set of procedure specifications, which define the operations on objects of the type, and by induction, the set of possible values for objects of the type.

2.2 A Model

To distinguish between various notions of correctness, a model of parallel executions is needed. We present an *object-based* model in which a set of *threads* (light-weight processes with shared address space) apply procedures to individual objects. The model, which is adapted from [HW90], is simple, yet rich enough to support the definition of most correctness notions. By object-based we mean that the state of a program consists of a set of abstract objects; each data type defines a class of objects, and that each operation is associated with a type and modifies at most a single object of that type. By convention, the first argument to each operation is the one that may be modified. The restriction that operations modify at most one object is probably too severe in general, but is met by the examples in this thesis. In the discussion below, we comment on the reasons for the restriction.

An operation is represented by an invocation event, occurring at the instant the procedure call is made, and a response event, occurring when the procedure returns. While procedure applications can overlap in time, we assume that the invocation and response events are totally ordered.

A *history* is a possibly infinite sequence of invocation and response events. An *invocation event* of procedure P on object x by thread T is written $\langle P(x, v_1, ..., v_n), T \rangle$, where $v_1, ..., v_n$ is the (possibly empty) list of the arguments (actuals) to P. Procedures may terminate either by returning or by signaling an exception. A *response event* of procedure P for thread T is written

$\langle R(v_1, ..., v_n), T\rangle$, where R is a termination condition of P and $v_1, ..., v_n$ is a possibly empty list of return values. (A *termination condition* is either a signal name or the special name *rtn*, which denotes a normal return.)

A *sequential* history is an alternating sequence of invocation and response events, starting with an invocation event. Given a history H and thread T, the *thread subhistory of H at T*, written $H|T$, is the subsequence of events in H that contain T. A history H is *well-formed* if for every thread T, $H|T$ is sequential. We consider only well-formed histories here, since the histories generated from actual programs are always well-formed. An invocation event i in history H *corresponds to* an event r if both contain the same thread T, i occurs before r, and no other event by T occurs between them. (By well-formedness, when r exists it must be a response event.) If H is a history and x is an object, the *object subhistory of H at x*, written $H|x$, is the subsequence of invocations events containing x (naming x as the first argument) and their corresponding response events.

Given a history H, an *operation* is a pair of events $\langle i, r\rangle$ such that i is an invocation event and r is the corresponding response event in H. An operation is the formal representation of a procedure application. An operation consisting of invocation $\langle P(x, v_1, ..., v_n), T\rangle$ and response $\langle R(w_1, ..., w_n), T\rangle$ is written more concisely as $\langle P(x, v_1, ..., v_n)/R(w_1, ..., w_n), T\rangle$, and when T is clear from context, both T and the surrounding angle brackets are dropped. When an invocation event in a history H has no corresponding response event, it is called a *pending invocation.*

An operation o_1 is said to *precede* another o_2 in history H if and only if the response event of o_1 occurs before the invocation event of o_2. This defines an irreflexive partial order, denoted $o_1 \prec^H_{g.t.} o_2$, and is called the *global time order of H*. The *program order of H* ($\prec^H_{p.o.}$) on operations is the global time order restricted to pairs of operations of the same thread, i.e., if $o_1 \prec^H_{g.t.} o_2$ and both o_1 and o_2 contain the same thread then $o_1 \prec^H_{p.o.} o_2$. Note that any two operations of the same thread are comparable in both orderings, because thread subhistories are sequential.

To relate sequential histories to specifications, we associate a sequence of states with the sequence of events, and use the state to assign a value each object named in the history. A sequential history $\langle i_1, r_1\rangle, \langle i_2, r_2\rangle, ...$ is *legal* if there exists a sequence of states $s_0, s_1, ...$ such that for each operation, $\langle i_k, r_k\rangle$ of procedure P_k, the pair of states $\langle s_{k-1}, s_k\rangle$ is a legal execution

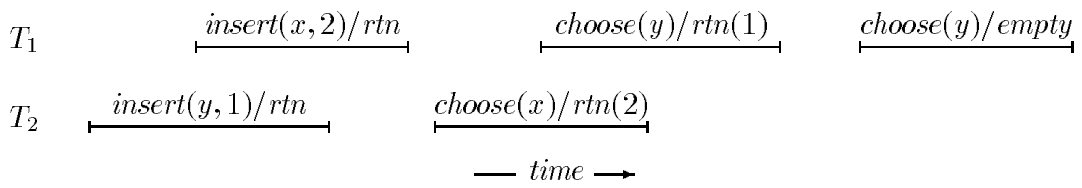

Figure 2-2: An Example History

of P_k.

The history model is precise, but pictures are more convenient for presenting the examples. Figure 2-2 gives a pictorial view of a history in which threads T_1 and T_2 are accessing two shared containers of integers, x and y, which are initially empty. The horizontal axis is time, and points on the vertical axis are threads. The duration of an operation is represented by a line segment (where the endpoints mark the time of invocation and response events of the formal model). The history is not sequential because operations of T_1 and T_2 overlap in time.

2.3 Strong Correctness

We begin by presenting two correctness notions from the literature, sequential consistency [Lam79] and linearizability [HW90]. These definitions extend the notion of correct sequential executions, which were called legal executions in the sequential case, to parallel executions. We classify them as *strong* notions because they place no restrictions on the use of operations; an implementation must behave according to its specification in an environment with arbitrary concurrency. Sequential consistency is the correctness notion assumed on memory operations in Chapter 3, while linearizability, as extended in Section 2.4, is the correctness notion for all data types other than shared memory. We justify this lack of uniformity in the discussion following the definitions.

Sequential Consistency

The notion of sequential consistency was proposed in [Lam79] as a requirement on shared memory for multiprocessors. It depends on the following definition of equivalence for histories.

Definition. Two histories H_1 and H_2 are *equivalent* if for every thread T, $H_1|T = H_2|T$.

When a history contains pending invocations, a complication arises in defining correctness. In some cases the effect of a pending invocation has been observed by other threads in the history. For example, a concurrent *choose* operation may return an element of a concurrent *insert* operation, even though the *insert* has not yet returned. To handle pending invocations, we use the following: if H' can be built from H by appending a finite number of response events, then H' is a *response extension* of H. It is not always possible to complete the pending invocations in this manner when the operations have waiting specifications, since the pending invocation may have no legal response event. For example, if a history ends with a pending *waiting_choose* operation, but the *container* is empty, then there are no response events allowed by the specification of *waiting_choose*. In this case, the pending invocations are ignored: let *nonpending*(H) be the subsequence of H with all pending invocations removed.

Definition. H is *sequentially consistent* if there exists a response extension H_{res} of H and a legal sequential history H_{seq} such that H_{seq} is equivalent to *nonpending*(H_{res}).

Informally, sequential consistency says that operations must appear to take effect instantaneously in program order. For memory operations *read* and *write* with the obvious sequential specifications, requiring the appearance of instantaneous behavior means that a *read* will never see a half-written value, and two *writes* will not leave a memory cell with partial results of each. Requiring that operations take effect in program order means that if a thread writes the value "1" into a cell and then reads the cell, the value returned will be either "1" or something written later, rather than an earlier value.

For pedagogical reasons, we also define local sequential consistency, which requires that individual objects are sequentially consistent.

Definition. H is *locally sequentially consistent* if for each object x, $H|x$ is sequentially consistent.

Any history that is sequentially consistent is also locally sequentially consistent. However, examples given in [HW90] and below demonstrate that the converse is not true, so local sequential consistency is strictly weaker than sequential consistency.

Linearizability

Linearizability is stronger than sequential consistency. It was proposed by Herlihy and Wing [HW90] as a correctness criterion for abstract data types in parallel programs, and Lamport used it as a condition on shared memory registers, which were called "atomic" registers [Lam80].

Definition. H is *linearizable* if there exists a response extension H_{res} of H, and a legal sequential history H_{seq}, such that H_{seq} is equivalent to $nonpending(H_{res})$ and $\prec_{g.t.}^{H_{res}} \subseteq \prec_{g.t.}^{H_{seq}}$.

As with sequential consistency, linearizability says that operations must appear to take effect instantaneously in program order, but it has the additional requirement that order of operations in the sequential history must be consistent with their order in the concurrent one. In effect, this means that each operation's instant (the point at which it appears to take effect), must be sometime between its invocation and response. An operation cannot take effect before the user has called the procedure, or after the procedure has returned control to the caller. Two operations that overlap in time can be "linearized" in either order, but ones that do not overlap cannot be reordered.

Both correctness notions have analogs in the theory for distributed transactions. If each transaction is equated with a single operation in our model, sequential consistency corresponds to serializability and linearizability corresponds to the histories in class Q [Pap79]. The fundamental difference between the two problem domains is that transactions are generally an arbitrary sequence of operations terminated by a special commit operation. Serialization is done on an entire transaction (i.e., a sequence of operations) rather than on individual operations. Protocols for serializability are typically not practical in a multiprocessor environment because they do not allow sufficient concurrency between individual operations on the same object.

Sequential consistency and linearizability are both extended from a single history to an implementation by universally quantifying over histories. An implementation of a data type specification S produces a history when it is executed. Such an implementation is *sequentially consistent* (*linearizable*) if for all histories H that it can produce, H is sequentially consistent (linearizable). It does not make sense to say that a single operation is sequentially consistent or linearizable without saying what other operations may appear in a history with it. However,

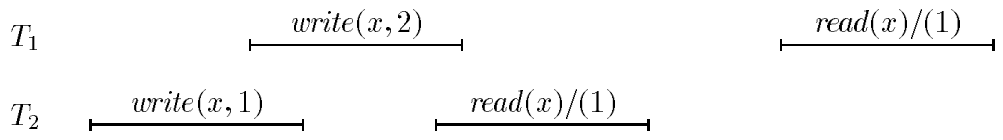

Figure 2-3: Linearizable history on shared location x

when the set of operations of interest is clear from context, because we know the type of an object for example, then we sometimes talk about the correctness properties of a particular operation.

Sequential Consistency vs. Linearizability

Using shared memory operations as an example, we compare sequential consistency and linearizability. For convenience, we assume that all locations are initially zero, and since neither *read* nor *write* raises an exception, we omit the termination condition, *rtn*, from the operations.

Even with the stronger notion of linearizability and deterministic specifications, the behavior of a concurrent execution is nondeterministic, because two overlapping operations can be linearized in either order. For example, assuming the variable x is initially 0, the history in Figure 2-3 is linearizable with T_1's *write* taking effect before the overlapping *write* of T_2. However, the history would also be linearizable if T_2's *read* returned 2; in that case T_1's *write* takes effect before T_2's. A variation that would not be linearizable is the one in which T_1 reads 1 and T_2 reads 2.

Linearizability is arguably stronger than necessary. It requires consistency with the real time ordering (i.e., "wall clock" time), which can be seen only by an omniscient observer, not by a process within the system. Consider, for example, the history in Figure 2-4. It is not linearizable because T_1's *read* returns the initial value, 0, even though it started after T_2's *write* had finished. However, if these are the only events in the system, then the programmer who wrote the program producing this history could not have known that T_1's *read* would happen after T_2's *write*: there is no synchronization or other communication that prevented T_1 from running faster and invoking the *read* earlier. Thus, from the user's perspective, the history is acceptable; the memory operations are behaving one would expect.

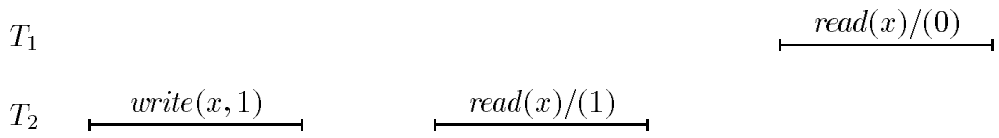

Figure 2-4: History that is sequentially consistent, but not linearizable

The advantage to linearizability over sequential consistency is that a linearizable system of objects can be built one object at a time, rather than being built as an entire system. The property of linearizability that makes this possible is called *compositionality* and given in Theorem 1. The proof of this theorem was presented by Herlihy and Wing [HW90], although the result was shown in a more abstract framework by Lamport [Lam80].

Theorem 1 [HW90] *H is linearizable iff for all objects x, $H|x$ is linearizable.*

Thus, implementations of linearizable objects can be composed and generic object implementations for common data types can be reused between different applications. For example, given a linearizable implementation of a *container*, one can build a state containing two instances of *containers*, and know that histories of multiple threads accessing the two containers will be linearizable. Or, given a linearizable implementation of a *container* and a linearizable implementation of a *stack*, histories that access *container* objects and *stack* objects will be linearizable. Furthermore, since object implementations are reusable, libraries of such implementations could be a basis for large-scale software development.

Sequential consistency is not compositional, because a history that is sequentially consistent at each object is not necessarily sequentially consistent at the global level. We demonstrate this point by the shared memory example in Figure 2-5. History H_1 is sequentially consistent with the *read* by T_1 linearized before the *write* by T_2. H_2 acts on shared object y rather than x, but is otherwise the mirror image of H_1, and is also sequentially consistent. History H_3 is not sequentially consistent, because the two *reads* cannot both be sequentialized ahead of the *writes* unless the program order is violated. H_3 is, however, locally sequentially consistent, since each of its objects subhistories are sequentially consistent. Local sequential consistency would be the property viewed by the user of a system in which each object is sequentially consistent.

Compositionality of modules is important in building large software systems, but is not

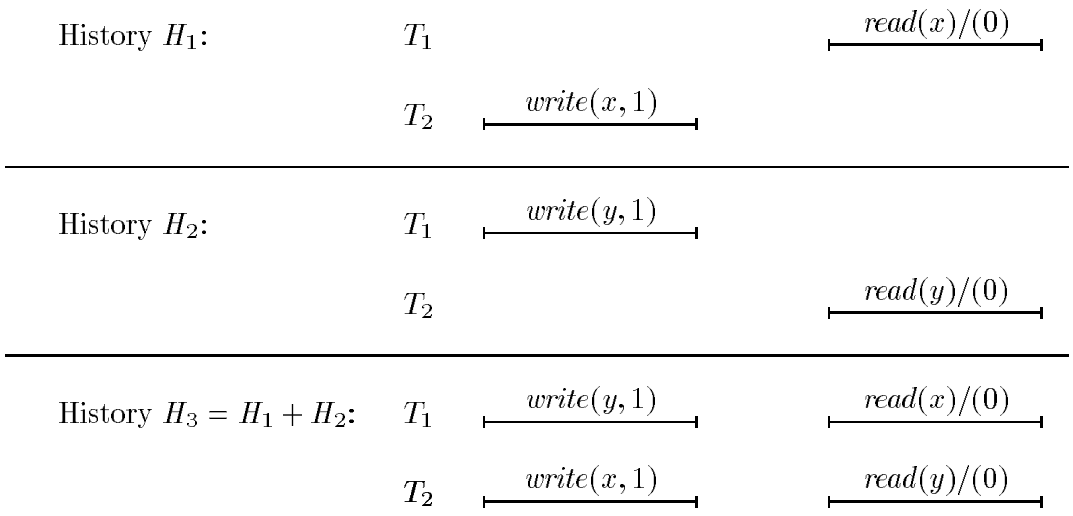

Figure 2-5: Sequential consistency is not compositional

used when a complete system is designed and implemented as a single unit. Given a system of sequentially consistent operations, once cannot write a program using only those operations that will determine whether or not the system is linearizable; it is only in composing subsystems that linearizability becomes important. A shared memory system, for example, is built as a complete system, not as a collection of individual memory cells, so sequential consistency is adequate as a correctness condition for shared memory. Furthermore, there appears to be a performance penalty to using linearizability rather than sequential consistency, since a straightforward way of turning a sequentially consistent implementation into a linearizable one is to add delays, making each operation wait until the effect of the operation is observable by every other thread. Attiya and Welch [AW91] prove that linearizable memory is, for a particular abstract machine model, more expensive than sequentially consistent memory.

For these reasons we assume that a machine provides sequentially consistent built-in operations, but require that software operations built on top are linearizable. The assumption on machine operations covers not only the usual *read* and *write* operations, but also basic synchronization operations, such as *test&set*, and thread manipulation operations, such as *fork* and *join*. Given a set of sequentially consistent primitives on which linearizable operations are

built, the low level primitives can be mixed with a higher level linearizable operations, and the resulting system will be sequentially consistent.

Before proceeding with variations of strong correctness, we remark on the underlying assumption of our object-based model that each operation acts on at most one object.[3] This assumption is crucial to the compositionality of linearizability, because Theorem 1 depends on object subhistories being pairwise disjoint, a property that would not hold if operations could act on more than one object. As we mentioned earlier, the assumption holds for the examples in this thesis, but the loss in expressive power that enables composition may not be a good trade-off in general.

2.4 Conditional Correctness

There are many cases in which parallel code does not depend on the full power of sequential consistency or linearizability, because all accesses to shared objects are done within critical regions. In this case, the programmer is relying on the linearizability of synchronization operations to guarantee mutual exclusion. A conditional notion of correctness based on this idea can allow more efficient implementations of the lower level abstractions. Conditional correctness notions have appeared in the multiprocessor literature for implementations of weak shared memory, where memory is shown to be sequentially consistent given certain assumptions about how memory operations are used [DS88, AH90b, SS88]. We generalize these concepts to arbitrary shared data types and give a linguistic mechanism for specifying the restrictions on use.

In this section we assume that the user of an abstraction is willing to make promises about how the operations will be used. Such promises are not new to parallel programs. In sequential programs, the *requires* clause on procedures constitutes such a promise; it states assumptions about the state in which a procedure will be invoked. In parallel programs, the promise will be an *interference specification*, which is an agreement between user and implementor that a certain set of operations will not be invoked concurrently.

An interference specification is an addition to a data type specification, indicating that certain pairs of operations interfere, and therefore should not be executed concurrently. It

[3] By "act on" we mean that the operation either mutates or observes a mutable object.

is the responsibility of the user of an object to ensure that no interfering operations are invoked concurrently. If the user fails to ensure this *non-interference*, then the behavior of the implementation is unconstrained. In general, an agreement in the form of a synchronization convention is needed between implementors of all threads in order to ensure non-interference. For example, a procedure P will be applied to object x only when holding a particular *lock* y. In this case, interference is prevented between two invocations of P by using a linearizable *lock* object y. (We include a specification of *locks* in Figure 2-6 along with the detailed examples that use *locks*.) To ensure non-interference we combine the interfering operations with operations that have no interference, such as the *acquire* and *release* operations on *locks*.

Formally, our model is extended to include a notion of interfering operations. With each data type D specified in a fixed set of data type specifications S, associate a binary relation I_D on invocation events. I_D is called an *interference relation* and is subject to the following constraints:

1. $(i_1, i_2) \in I_D$ implies $(i_2, i_1) \in I_D$. (Interference relations are symmetric.)

2. $(\langle P_1(x, v_1, ..., v_n), T_1\rangle, \langle P_2(y, z_1, ..., z_m), T_2\rangle) \in I_D$ implies $x = y$. (Procedures only interfere when applied to the same object.)

3. $(\langle P_1(x, v_1, ..., v_n), T_1\rangle, \langle P_2(x, z_1, ..., z_m), T_2\rangle) \in I_D$ implies x is of type D. (The interference relation for D only constrains objects of type D).

Note that an interference relation need not be transitive or reflexive. We refer to the union of the I_D's as the *interference part of* S, or simply as an *interference specification.*

Roughly speaking, a history contains interference if there are two overlapping operations that have interfering invocation events. The possibility of pending invocations complicates the precise statement of interference, since pending invocations are technically not operations. Let S be a set of data type specifications with interference part I, and let H be a (possibly infinite) history of operations specified in S.

Definition. A history H *contains interference* if there are two distinct invocation events i_1, i_2 in H such that $(i_1, i_2) \in I$ and one of the following is true:

1. both i_1 and i_2 are pending, or

2. i_1 has a corresponding response event r_1, and $i_1 \prec^H_{g.t.} i_2 \prec^H_{g.t.} r_1$.

A history that does not contain interference is *interference free*.

We now give the main definition of this section, which is a conditional form of linearizability.

Definition. H is *linearizable modulo interference specification* I if either H is linearizable or H contains interference.

Linearizability modulo an interference specification is weaker than linearizability, although when the interference specification is empty, the two notions are equivalent. The interference specification acts as an additional contract between the user and the implementor of a set of objects: the user must ensure that histories are interference-free, and the implementor must (conditionally) ensure linearizability. Henceforth, when the existence of an interference specification is clear from context, we simply use linearizability to mean linearizability modulo the interference specification.

The compositionality of linearizability (Theorem 1) provides the user with means of ensuring that histories are interference-free. If synchronization objects are linearizable, then subhistories containing only those objects must be linearizable. The specification of a *lock* asserts that a *lock* can only be acquired when it isn't already held, so if two threads attempt to acquire a *lock* simultaneously, only one may succeed at a time. The other thread's *acquire* operation will be linearized after the first one's *release*.

Although the general form of an interference specification is a relation on invocation events, which include operand values, many interesting interference specifications can be stated more simply as a relation on procedures. Consider the *container* example in Figure 2-1 with *insert* and *choose* operations. If a *print* operation is added to an existing implementation of *containers*, the *insert* and *choose* operations would probably have to be modified to synchronize with *print*, so that a consistent view of the object would be printed. The synchronization would add overhead, probably in the form of *lock* acquisition, to *insert* and *choose*. Some of this overhead would be incurred regardless of whether *print* was being used. A better solution is to leave synchronization to the user of the *container* by adding an interference specification stating that *print* interferes with both *insert* and *choose*.

To formally denote an interference specification of this kind, we use a *procedural interference*

relation R_D, which is a symmetric binary relation on the procedures defined on type D. We then extend R_D to an interference relation I_D as follows. Let (P_1, P_2) be a pair of procedures in R_D, where P_1 and P_2 may be the same procedure. If i_1 is an invocation event of P_1 on object x, and i_2 is an invocation event of P_2 on x, then (i_1, i_2) is in I_D; I_D contains all such pairs and nothing else. Procedures in R_D are said to *interfere*.

Some interference relations cannot be defined by a relation on procedures, because they depend on the argument values. In Chapter 3, we give an example of such an interference relation in the specification for a *mapping* type. We describe the interference relations for the *mapping* implementation directly, and demonstrate that it is not an extension of a relation on procedures.

2.5 A Liveness Property

Linearizability, sequential consistency, and all their variants are *safety* properties; they make a statement about what bad things cannot happen in an execution. More precisely, safety properties are *prefix closed*: a safety property that holds on history H also holds on any prefix of H. A *liveness* property, on the other hand, makes a statement about what good things must eventually happen. For example, a procedure must eventually terminate; a scheduler must eventually allow every thread to run; a semaphore must eventually be granted to a waiting thread.

Liveness properties can be formalized in a temporal logic, of which there are many examples, including Lamport's Temporal Logic of Actions, which was designed to be simple and usable in real world examples [Lam90]. In Chapters 4 and 5, where entire parallel programs are described at an abstract level, some examples of application-specific liveness properties are informally stated. For the concurrent objects discussed here and implemented in Chapter 3, we choose a particular liveness property, *non-stopping*, for all data types. This limits the class of implementations that are considered correct, but the programming model is simplified by having a single liveness property for all data types.

Roughly, *non-stopping* means that in any execution in which some procedure is allowed to terminate, some procedure must eventually terminate. The meaning of "allowed" depends on the set of specifications, as well as the safety part of the correctness condition on data types.

lock = **datatype has** create, acquire, release

create = **procedure** () **returns** (lock)
ensures: returns a new (unlocked) lock

acquire = **procedure** (l : lock)
when: l is unlocked
ensures: l becomes locked

release = **procedure** (l : lock)
ensures: l becomes unlocked

end lock

Figure 2-6: Specification of locking mechanism

In the following discussion we use linearizability (possibly modulo an interference specification) as the safety property.

Recall from Section 2.1 that a procedure specification is *waiting* if there are states in which the procedure is not allowed to terminate (or make any other observable state change). One common example of a waiting specification is the specification of a *acquire* primitive on *locks*; Figure 2-6 gives a specification of the *lock* data type. The *acquire* procedure is not allowed to return until it detects a state in which the *lock* object x is available. In defining a liveness property, we distinguish between a specification that requires waiting, and an implementation that waits when the specification does not mandate it. A non-stopping data type implementation is one that waits only when its specification requires that it wait. Thus, the definition of non-stopping, which is the condition on implementations, depends on the notion of non-waiting, which is a condition on specifications. Examples of stopping implementations are procedures that loop forever, and procedures that deadlock when invoked concurrently; from the users perspective these are two cases are indistinguishable, since in both cases the procedure appears to stops before returning. We require that all operations on a concurrent object are non-stopping. The analogous condition from sequential programs is that all operations are implicitly required to terminate.

While procedures that stop arbitrarily are unusable, procedures that stop on the *when* clause of a waiting specifications can be quite useful. So far, we have seen two examples of waiting procedure specifications: the *acquire* procedure and the *waiting_choose* procedure

for *containers*. Chapter 3 gives a number of object implementations, but the only one with a waiting procedure is the *lock* type. We argue that non-waiting procedures are easier to schedule efficiently, and therefore limit waiting specifications to synchronization objects. A history containing only operations that non-waiting (as well as non-stopping) has the nice property that all the operations will eventually terminate, once the user stops invoking new operations.

In the restricted case that all specifications are non-waiting, the non-stopping condition means that every finite execution is free of pending invocations, i.e., every operation eventually terminates.

To define non-stopping, we make use of the following notion of *response extendible*; it characterizes those histories having at least one pending procedure invocation that could be completed with a response event. Let $H \cdot r$ denote the history formed by appending event r to the end of H.

Definition. A finite history H is *response extendible* if there exists a response event r, such that $H \cdot r$ is well-formed and linearizable.

A history that is not response extendible is called *unextendible*. Note that a history without pending invocations is unextendible, although the existence of pending invocations is not sufficient to make a history extendible. Figure 2-7 gives an example of a history that has a pending invocation but is unextendible. There is a single pending invocation, the invocation of *acquire* by T_2, and the specification of *acquire* prevents it from returning while z is held by T_1. The inability to extend a history is a compositional notion: H is unextendible if and only if $H|x$ is unextendible for all objects x in H.

The notion of response extendible is used to define non-stopping.

Definition. A history H is *stopped* if it is finite and response extendible.

An implementation that produces no stopped histories is *non-stopping*. The history in Figure 2-8 is stopped, while the history in Figure 2-7 is not stopped. Both histories are finite, and both have a pending invocation events, but only the history in Figure 2-8, in which T_1 has released the *lock* z, is response extendible. We remark that the important direction of compositionality holds for the non-stopping condition: if for all objects x in H, $H|x$ is not stopped, then H is

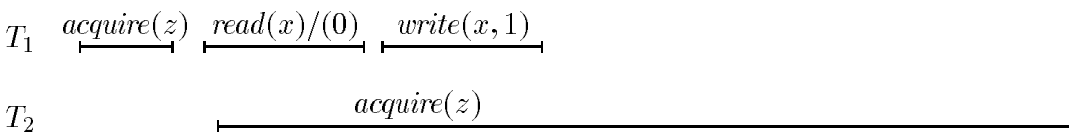

Figure 2-7: An unextendible history

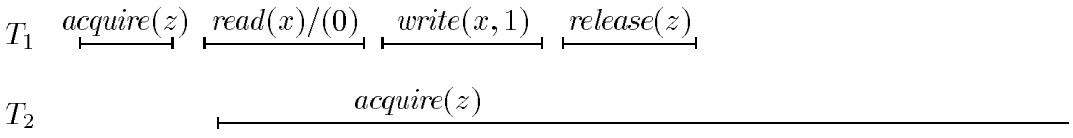

Figure 2-8: A stopped history

not stopped. The converse is not true, i.e., H may be stopped at some objects and still be an infinite execution.

Our notion of non-stopping is weaker than two other liveness notions for concurrent objects: non-blocking and wait-free [Her88, Her90]. Both of those notions require forward progress in the presence of failures, whereas our model of computation does not admit failures. Typically, implementing data types to resist failure requires additional overhead, both in programming difficulty and in performance.

2.6 Discussion

In this chapter we defined two new correctness notions for concurrent data types: linearizability modulo an interference specification (a safety property), and non-stopping (a liveness property). All of the data types that are specified and implemented in this thesis will be linearizable, sometimes with interference specifications. Most of the implementations in Chapter 3 are

non-stopping, although in one instance it is only non stopping with high probability. The example implementations in Chapter 3 provide evidence for the practicality of these notions: the correctness conditions are strong enough to allow modular design and reuse of program components, but are weak enough that high performance programs can result.

The notion of interference between operations has been exploited in various ways in related work. As noted earlier, some multiprocessor memory implementations make use of interference information to schedule the operations. In particular, the algorithm of [SS88] schedules memory operations so that memory that is not sequentially consistent at the hardware level appears sequentially consistent to the programmer. The algorithm requires data flow analysis to determine which interfering operations may actually execute concurrently, which may limit its practicality. The FX programming language [GJLS87] uses a kind of interference specification on procedures to determine whether two procedures may be executed in parallel. The approach in FX is conservative–two procedures are not executed concurrently if they contain interfering memory operations. All of these approaches use interference information at the level of memory operations, whereas our interference specifications are given at the abstract level. The reason for using an abstract condition like linearizability is to allow concurrency between implementations that interfere at the low level, but still compute a correct answer at the abstract level.

Linearizability would be acceptable from our point of view as a correctness condition on shared memory. However, we only rely on sequential consistency of memory operations, including *reads*, *writes* and any synchronization primitives provided by the architecture. Although many shared memory implementations are actually linearizable, there are proposals for sequentially consistent memory that is not linearizable, so we used the weaker assumption that memory is sequentially consistent. See [AH90a, GLL+90, BR90, DS88, SS88] for examples of memory implementations that are sequentially consistent but not necessarily linearizable.

Chapter 3

Implementing Concurrent Data Types

In this chapter we present specifications and implementations of a number of concurrent data types. Each of the types is used in the implementation of either matching (Chapter 4) or completion (Chapter 5). The examples are also interesting in their own right for a number of reasons:

- Some of the interfaces involve interference specifications, thereby demonstrating the practical importance of the ideas presented in Chapter 2.
- Some of our implementations perform dynamic memory allocation, which is a problem that is often ignored in examples from the literature. Dynamic allocation is essential in symbolic programs, and can be difficult to do correctly in a parallel environment. (We do not completely solve the dynamic object problem, as our implementations do not perform either deallocation or garbage collection.)
- The code makes use of sequentially consistent shared memory, often without the use of explicit synchronization primitives. This demonstrates the practical power of sequentially consistent memory, as opposed to weaker models of memory that assume the programmer will use synchronization around all shared memory accesses [DS88, AH90b].

- The implementation of the *mapping* type uses a novel backoff strategy for avoiding collisions in the space of keys being mapped.

- Taken as a whole, the set of data types demonstrate the power of a simple class of abstractions. All implementations are non-stopping and linearizable modulo an interference specification, and with the exception of a small set of synchronization primitives, all procedures have non-waiting specifications. This is in contrast to richer process-oriented programming models.

Throughout the chapter we also give general techniques for implementing concurrent objects. For example, the announcement board structure that is used in the *mapping* implementation appears in some of Herlihy's wait-free algorithms [Her90], and the algorithm used for assignment generalizes Dijkstra's mutual exclusion algorithm [Dij65]. Some of the problems that arise in the examples are also general. The need for thread-specific data within objects can reduce contention and improve locality, but it complicates the interface of the object. The most common example of an object with thread-specific data is a pool of free memory, where separate memory pools are used to reduce contention, but a global pool is used to refill local pools. We refer to objects with thread-specific data as *multi-ported objects*, since each thread has a unique instance of (or pointer to) the object, but all the instances behave semantically as a single object.

The presentation in this chapter is bottom-up in that the later implementations make use of the earlier ones. Before presenting the individual data types, however, we discuss in Section 3.1 some of the design decisions that pervade the implementations in this chapter. Section 3.2 gives interface specifications for the lowest level abstractions: memory, synchronization primitives, and counters. Section 3.3 gives some examples of more interesting concurrent objects: queues, dynamic arrays, and mappings.

3.1 Design Decisions

In Chapter 2 we presented an approach to writing specifications, and we defined correctness notions for implementations of data types. In implementing the data types used in our programs, we test the expressiveness of the specifications and the generality of the correctness notions.

We summarize the correctness conditions here, and comment on some of the design constraints implied by the correctness conditions.

- All data type operations are linearizable. Each operation must appear to take affect instantaneously, although concurrency can (and does) exist in our implementations.
- The object-based model of Chapter 2 is assumed. Given a data type D, each operation on D takes a single object of type D, plus some number of values, and modifies at most the one object.
- With the exception of the *acquire* procedure on *locks*, all procedure specifications are non-waiting. This simplifies scheduling, since a procedure that cannot make progress simply returns, rather than having to be de-scheduled.
- Each data type is non-stopping. This is a relatively weak liveness requirement that was chosen to keep overhead low; it prohibits deadlock, but permits a form of livelock that, with care, can be avoided by the user.
- Interference specifications are allowed. This is done to allow for simpler and more efficient implementations at some cost to the simplicity of interfaces.

These design constraints are specific to our approach to concurrent data type specification and implementation, but they do not directly aid or inhibit use of the transition-based approach in Chapter 4. However, the high-level approach does influence the low-level implementations in one regard: we assume there is application-specific scheduling, which means the application program schedules tasks (short-lived computations) on top of the system-level threads (long-lived computations) [RV89, CG89, JW90]. The scheduling model will be discussed further in Chapter 4, but for now, the important issue is the interaction between scheduling and synchronization.

In general, a task that is forced wait for a synchronization event may either busy-wait, i.e., spin on a synchronization condition, or it may block, i.e., notify the scheduler that it is waiting and should therefore be de-scheduled. When blocking synchronization is used with application-specific scheduling, the operating system scheduler can be notified directly, or the blocking mechanism can first go through the application scheduler. In either case the price

of a context switched is paid if the synchronization condition does not hold. When spinning synchronization is used, neither scheduler is notified, so the overhead of communication with the scheduler and of possible context switching is avoided. We choose to avoid all communication with the operating system scheduler, and build spinning synchronization primitives directly from hardware primitives such as *test&set*.

There are two potential performance problems with spinning synchronization primitives: spinning can waste valuable resources if the synchronization condition is not satisfied quickly; and spinning interacts badly with operating system events such as paging, because the operating system has no information about synchronization events. We address each of these concerns in turn.

The first problem with spinning synchronization primitives is that applications with long and frequent critical regions will waste processor resources. We therefore divide synchronization conditions into *short-term* and *long-term*, and treat the two cases separately.

- A short-term condition is one that is likely to be satisfied within a few instruction cycles, e.g., entry into a critical region that contains a small fixed number of instructions to be executed. Short-term conditions are handled by spinning synchronization.

- A long-term condition is one for which the bound on waiting time is known to be long or is difficult to predict, e.g., waiting for the result of another task's computation. Long-term conditions are handled by "aborting" the computation and returning to the caller.

In the case of a long-term condition, it is up to the caller to determine whether the operation should be retried, or whether the caller's own computation should be aborted. A chain of aborted computations will eventually reach the application scheduler, thereby allowing a different task to be scheduled. The effect of this chain of aborts is similar to having a lower-level operation notify the application scheduler directly that its task should be de-scheduled, but scheduler code is greatly simplified by avoiding this sort of de-scheduling. In particular, the scheduler does not have to manage task contexts for partially executed tasks. Note that our use of the word "abort" does not imply that an aborting procedure leaves all objects unchanged, as if it had never been invoked, but that the possibility of aborting is part of the procedure's specification. An aborting procedure must leave objects in a consistent state so that future op-

erations will behave correctly; unlike a transaction system, ours does nothing to automatically ensure consistency of objects on aborts.

The second problem with using spinlocks is that, because the operating system is not notified of synchronization events, it may de-schedule a thread while it is holding a lock, thereby preventing other tasks from entering a critical region protected by that lock. We use only as many threads as there are available processors, in an attempt to reduce the frequency of de-scheduling by the operating system, but because we do not prevent such de-scheduling entirely, application performance may be erratic. A more complete solution to the problem of operating system interaction is given in recent work by Anderson and others [ABLL90]. They modify the operating system interface to allow the application scheduler to communicate with the operating system scheduler, making it possible to maintain the invariant that the number of threads is no greater than the number of available processors. Our approach to building applications would well-suited to their two-level model of scheduling.

3.2 Specifications of the Hardware Abstractions

We start by specifying the abstractions provided by the Firefly hardware [TSJ87]. The Firefly has coherent caches that implement sequentially consistent memory.[1] In addition to *read* and *write* operations, the hardware also has a small set of *interlocked* instructions, instructions that are indivisible with respect to each other. The interlocked instructions include a *test&set* primitive, as well as an *add* primitive. Although the interlocked instructions and memory operations all act on memory locations, we model the memory system as having three different types of objects: *locations*, *spinlocks*, and *accumulators*. *Locations* are memory cells with *read* and *write* operations, while *spinlocks* and *accumulators* have a *read* operation and one of the interlocked instructions. Modeling memory by three different types of objects is cleaner than using a single memory cell type with the union of the operations, because the built-in *write* operation interferes with some of the interlocked instructions. We discuss this point further in the sections on *spinlocks* and *accumulators*.

[1]The memory abstraction actually meets the stronger requirement of linearizability, but as discussed in Chapter2, we only depend on sequential consistency.

location = **datatype** [*t*: **type**] **has** create, read, write

create = **procedure** () **returns** (location)
ensures: returns a new uninitialized location

read = **procedure** (*l*: location) **returns** (*t*)
ensures: returns the value stored at *l*.

write = **procedure** (*l*: location, *v*: *t*)
ensures: *v* is stored in *l*.

end location

Figure 3-1: Specification of *locations*

3.2.1 Memory Used as Locations

The Firefly provides coherent shared memory, which has the specification given in Figure 3-1. For uniformity, we refer to *create*, *read*, and *write*, and as procedures, although the procedure names will not be used explicitly in our programs. Instead, a *read* operation is implicit in any reference to a shared location within an expression, and a *write* is implicit when a shared location is named on the left-hand side of an assignment statement. Note that *read* and *write* operations involve two different types, so in expressions with location objects there is an implicit type coercion between a *location* and value of type *t*. The effect of the *create* operation is obtained in the code by allocating a contiguous sequence of shared memory locations.

3.2.2 Memory Used as Accumulators

Various kind of counters are useful for building higher level abstractions. In addition to the uses that are prevalent in sequential programs, parallel programs often use counters for synchronization. Counters require some synchronization within their implementation. The obvious implementation of an *add* operation, in which a shared counter is read into a local register, incremented, and then written back does not work in parallel, because two concurrent *adds* can read the same value. We make use of two different kinds of counters in our programs, one called an accumulator (*accum*), and the other simply called a *counter*. An accumulator abstraction is provided by the hardware and is specified here, while the *counter* abstractions is implemented in software and is specified in Section 3.3.

```
accum = datatype has create, read, add
    create = procedure () returns (accum)
        ensures: returns a new accumulator set to 0
    read = procedure (a: accum) returns (int)
        ensures: returns the value of a
    add = procedure (a: accum, i: int)
        ensures: adds i to a
    end accum
```

Figure 3-2: Specification of *accums*

A specification for the *accum* type is given in Figure 3-2. Note that the *add* operation changes the value of an accumulator without returning its value. This is an important limitation of accumulators, since it implies that accumulators cannot be used to generate unique integers for concurrent threads, a synchronization technique used in the *queue* implementation below. Nevertheless, accumulators are sufficient for other kinds of synchronization: the matching program in Chapter 4, for example, uses accumulators for termination detection.

An *accum* is actually just a memory location, and the *read* on *accums* is the same hardware *read* that is used on *locations*. The *add* operation is an interlocked instruction provided by the Firefly hardware, but because of the manner in which interlocked instructions are implemented, the *write* operation cannot be used on *accums*. All interlocked instructions are implemented by a single hardware lock, the *interlock lock*, which is separate from the arbitration hardware on the memory bus. Before a processor executes an interlocked instruction it must acquire the interlock lock; once acquired, the processor executes the interlocked instruction and then releases the interlock lock. In the case of the (interlocked) *add* instruction, the hardware performs a *read*, followed by an addition, followed by a *write*, with all three steps done inside the critical section enforced by the interlock lock. *Write* instructions do not acquire the interlock lock, so if an *add* is executed concurrently with a *write*, the *write* may be lost. The following execution illustrates the problem. Processor P_1 is executing an *add* of i to shared variable x, which has the initial value v_1, and processor P_2 is executing a *write* on x with the value v_2. The interleaving of hardware operations is:

1. P_1 acquires the interlock lock.

2. P_1 *reads* the initial value of x, which is v_1.
3. P_2 *writes* writes the value v_2 into x.
4. P_1 increments v_1 by the given integer i and writes the result $v_1 + i$ into x.

The history is not sequentially consistent because the end result should have been either $v_2 + i$, as if the *write* had happened before the *add*, or v_2, as if the *write* had happened after the *add*. The hardware designers could have implemented *write* so that it did not interfere with *add*, but the cost in performance of *writes* might have made the memory system unusable. In Section 3.3 we give a software implementation of a *counter* type, which is similar to an *accum*, but has a *write* operation.

Since the problem with *write* and *add* is one of interference, it could be addressed by adding an interference specification to memory. Moreover, the same sort of interference appears with the *test&set* operation on *spinlocks*, described below, so all three memory abstractions could be combined into a single abstraction with an appropriate interference specification. In using memory, however, we always ensure that the interference specification is observed by designating each memory cell as either a *location*, an *accum*, or a *spinlock*, and consistently using it as such. It makes the higher level programs more readable if the set of allowed operations on a given memory cell is a function of its type name.

3.2.3 Memory Used for Spinlocks

Synchronization is sometimes done by waiting for a particular value to appear in a *location* or an *accum*, but in other cases it is convenient to use a read-modify-write primitive such as the interlocked instruction *test&set*. The *spinlock* data type specified in Figure 3-3 provides such an operation. Like accumulators, *spinlocks* are simply memory locations that are being used for interlocked instructions, and like the *add* operation on *accums*, the *test&set* operation on *spinlocks* interferes with *write*, so *write* is not an operation on *spinlocks*. The *test&set* operation atomically tests to see whether the *spinlock* is held, and if not, acquires it. The *reset* operation releases the *spinlock*.

The specification of *spinlocks* includes a *test* operation to read the value of a *spinlock* without modifying it. Our applications do not use *test*, but it is available through a memory *read* and can be useful implementing *test-and-test&set* style synchronization. See [And89] for a description

spinlock = **datatype has** create, test&set, reset

create = **procedure** () **returns** (spinlock)
ensures: returns an unlocked spinlock

test&set = **procedure** (*s*: spinlock) **returns** (bool)
ensures: if *s* is unlocked then lock *s* and return true
else leave *s* locked and return false

test = **procedure** (*s*: spinlock) **returns** (bool)
ensures: if *s* is unlocked then return true
else return false

reset = **procedure** (*s*: spinlock)
ensures: unlock *s*

end spinlock

Figure 3-3: Specification of *spinlocks*

of various strategies for using *spinlocks*, including *test-and-test&set*, and a discussion of the relative performance of the strategies.

3.3 Reusable Software Abstractions

In this section we describe some of the linearizable concurrent types that are used in the matching and completion programs. The first two abstractions, *locks* and *counters*, have no real concurrency, but are lightweight primitives that are used to minimize the overhead of synchronization in both applications. The *queue*, dynamic array (*dyn_array*), and *mapping* implementations have varying degrees of concurrency, and they perform dynamic memory allocation internally when necessary.

3.3.1 Locks

Locks provide a cleaner interface to synchronization than *spinlocks*. *Locks* can be implemented by either blocking or spinning without changing the specification. The *lock* specification was included as part of an example in Chapter 2, and it is repeated in Figure 3-4 for completeness.

Recall that synchronization primitives are the exception to the rule that procedures must have non-waiting specifications, and in particular, the *acquire* operation has a waiting specifi-

lock = **datatype has** create, acquire, release

create = **procedure** () **returns** (lock)
ensures: returns a new unlocked lock

acquire = **procedure** (*l*: lock)
when: *l* is unlocked
ensures: *l* becomes locked

release = **procedure** (*l*: lock)
ensures: *l* becomes unlocked

end lock

Figure 3-4: Specification of *locks*

cation. The *lock* specification is not safe, in that any thread may release a *lock* held by another thread; it is only programming convention that prevents this from happening. A safer locking abstraction could be defined [BGHL87], or, alternatively, programming language restrictions could enforce the programming convention that locks are always released by the thread that acquired them [Bir89].

The *locks* in our programs are implemented using *spinlocks*: the *acquire* operation repeatedly calls *test&set* until it successfully sets the *spinlock* and the *release* procedure calls *reset*. Our implementation is not fair, in the sense that one thread may be starved if other threads *acquire* and *release* the *lock* with high enough frequency. Since one of our goals in writing parallel programs is to avoid contention by using critical regions sparingly, we have never observed *lock* starvation in practice. As the number of processors grows, the possibility of starvation would also grow, however, it is likely that increased contention would force an alternate implementation before starvation became a problem.

3.3.2 Counters

A specification of the *counter* type is given in Figure 3-5. A *counter* is similar to an accumulator, but unlike the *add* operation on *accums*, *inc* returns the value of the *counter*. *Inc_if_less* is like *inc*, except that the increment is done conditionally, depending on whether the current value of the *counter* is less than the second argument. *Inc_if_less* must be part of the *counter* interface in order to be linearizable; it cannot be built out of *inc* and *read* without another level

counter = **datatype has** create, inc, inc_if_less, read, write

create = **procedure** (*i*: int) **returns** (counter)
ensures: returns a new counter set to *i*

read = **procedure** (*c*: counter) **returns** (int)
ensures: returns the value of *c*

write = **procedure** (*c*: counter, *i*: int)
ensures: assigns *i* to *c*

inc = **procedure** (*c*: counter) **returns** (int)
ensures: increments *c* by 1 and returns the old value of *c*

inc_if_less = **procedure** (*c*: counter, *i*: int) **returns** (int) **signals** (not_less)
ensures: if *c* is less than *i*,
then *c* is incremented and the old value of *c* returned,
otherwise *c* is not changed and not_less is signaled

end counter

Figure 3-5: Specification of *counters*

of locking around all the operations that will be used concurrently with it.

The observation that an indivisible *inc_if_less* cannot be built by simply composing a comparison with an increment has unfortunate implications for concurrent object reuse in general. A data type defines the set of operations that are linearizable, and to extend this set we may have to implement the entire type again. In sequential programs complex operations are built out of simpler ones using sequential composition, and while the same can be done in parallel programs, an operation built from sequential composition is not linearizable. Furthermore, if we attempt to implement very general linearizable data types that have many operations, performance will suffer. Returning to the example, the operations on a *counter* are a proper superset of the operations on *locations*, yet we would not want to replace *locations* by *counters* because the *counter* implementation, as described below, has an extra level of locking.

A *counter* is represented by a record that contains the integer value of the *counter* and a *lock*. With the exception of *read*, all *counter* operations are written using critical regions: they call *acquire*, do whatever computation is necessary on the value, and then call *release*. The *read* operation does not require locking, but the *write* implementation does; it is this locking that gives the *write* operation on *counters* its high overhead.

The *counter* implementation allows for concurrent use, but because of the critical regions in the implementation, there is no real concurrency, except with *reads*. Fortunately, each operation is fast, and the critical regions are all a bounded number of instructions. *Counters* have more synchronization overhead than *accums* since, even in the best case, all the mutating operations on *counters* require two interlocked instructions compared with only one for *accums*. This difference is probably not significant in the context of a larger program that uses such operations infrequently. Nevertheless, we use *accums* whenever possible to avoid the extra interlocked instructions.

Because all interlocked instructions on the Firefly use a single hardware lock, contention for the interlock lock can be a problem before contention for application-level locks or contention for the memory bus becomes a problem. In terms of the programming model, this means that a synchronization primitive such as *test&set* may be significantly more expensive than the two memory operations it contains.

3.3.3 Queues

In this section we give a specification and implementation of a *queue*. The programming language that is used throughout this chapter and Chapter 4 is a hybrid of existing programming languages, CLU [LAB+81] and C. From C we take explicit memory allocation, since dynamic memory allocation is an important feature of our *queue* implementation. We also take curly braces for grouping statements, since they save space on a written page. From CLU we take parameterized data types and a signaling mechanism. Recall from Chapter 2 that signals are simply alternate termination conditions for an operation, and that a signaling procedure cannot be resumed.

A specification for a *queue* is given in Figure 3-6. It includes an interference specification that says that the *print* procedure interferes with both *enqueue* and *dequeue*. This constrains the use of *queues*: executions that involve an invocation of *print* that is concurrent with *enqueue* or *dequeue* have unspecified behavior. Without the interference specification, *print* operations would have to work correctly while concurrent *enqueues* and *dequeues* were being done. This would complicate the implementation of *print*, but more importantly, would add complications and overhead to the implementations of *enqueue* and *dequeue*.

queue = **datatype** [*t*: **type**] **has** create, enqueue, dequeue, is_empty, print

interference between: (enqueue, print), (dequeue, print)

create = **procedure** () **returns** (queue)
ensures: returns an empty queue

enqueue = **procedure** (*q*: queue, *e*: *t*)
ensures: adds *e* to the head of *q*

dequeue = **procedure** (*q*: queue) **returns** (*t*)
ensures: if *q* is empty then signal empty
otherwise remove and return the tail of *q*

is_empty = **procedure** (*q*: queue) **returns** (bool)
ensures: returns true if *q* is empty, false otherwise

print = **procedure** (*q*: queue)
ensures prints the elements of *q* in order

Figure 3-6: Specification of *queues*

The *queue* specification is non-waiting. Its implementation, given in Figure 3-7, is linearizable and non-stopping. The non-stopping property, as discussed earlier, implies that the *queue* implementation does not deadlock. The figure does not include implementations of *is_empty* and *print*—both are straightforward.

A *queue* is represented by a contiguous block of *locations* (called a *static_array*), plus a number of pointers into the block. *High* points to the head of the queue, *low* points to the tail of the *queue*, and *defined_high* points to the head-most slot that has a valid value. *Enqueue* increments *high* and uses the return value of *counter$inc* as its unique slot in the *static_array*. After the increment, *defined_high* is less than *high*, but once the slot has been filled, *defined_high* is incremented. Similarly, *dequeue* increments *low* to get a unique slot from which to *read* a value.[2]

The *static_array* is created and expanded using the *alloc* and *realloc* operations. The *realloc* operation copies one block of memory into another after doing an allocation. To avoid having more than one copy available at any time, the *enqueue* ensures that *reallocs* are done sequentially. The *alloc* operation, on the other hand, is implemented to allow for concurrency.

[2]In the procedure headers, the type "cvt" converts between the abstract and representation types. The caller passes a *queue*, but the procedure bodies access the object as a *rep* type.

```
queue = cluster [t: type] has create, enqueue, dequeue, print
    rep = record[max_alloc, defined_high: location[int],
                 low, high: counter,
                 elts: static_array[t]]
    create = procedure () returns (cvt)
        q: rep := alloc(sizeof(rep))
        q.max_alloc := 0
        q.low := counter$create(0)
        q.high := counter$create(-1)
        q.defined_high := -1
        q.elts := null
        return(q)
      end create
    enqueue = procedure (q: cvt, e: t)
        i: int:= counter$inc(q.high)
        if i < q.max_alloc then
            {q.elts[i] := e
            while (q.defined_high) < i - 1 do {} % spin
            q.defined_high := q.defined_high + 1}
        else {while q.defined_high < i - 1 do {} % spin
            while i >= q.max_alloc do
                if i = q.max_alloc then
                    {if q.max_alloc = 0 then
                      {q.elts := alloc(min_queue_size)
                      q.max_alloc := min_queue_size}
                    else
                      {q.elts := realloc(q.elts, (q.max_alloc + 1) * qmult, q.max_alloc)
                      q.max_alloc := q.max_alloc * queue_mult}}
            q.elts[i] := e
            while q.defined_high < i - 1 do {} % spin
            q.defined_high := q.defined_high + 1}
      end enqueue
    dequeue = procedure (q: cvt) returns (t) signals (empty)
        h: int := counter$read(q.high) + 1
        max: int := q.max_alloc
        if counter$read(q.low) > counter$read(q.high) then signal empty
        i: int := counter$inc_if_less(q.low, min(h, max))
                  except when not_less: signal empty
        while q.defined_high < i - 1 do {} % spin
        return(q.elts[i])
      end dequeue
```

Figure 3-7: Implementation of *queues*

Although the details are not shown in the *queue* implementation, memory allocation is done by using separate pools for each thread, so there is potential for real concurrency between multiple allocations.

Correctness of the *queue* implementation depends on the following key invariants.

- $0 \leq \text{low} \leq \text{high} + 1$
- $-1 \leq \text{high} \leq \text{max\_alloc}$
- $-1 \leq \text{defined\_high} \leq \text{high}$
- $\text{size} = \text{high} - \text{low} + 1$, where $\text{size} = 0 \Rightarrow \text{is\_empty}$

The idea of using an indivisible *inc* operation (often called *fetch-and-add*) has appeared elsewhere [GLR83, HW90]. Our implementation is unique in two respects: space is allocated dynamically, and access to individual slots in the queue is controlled by a pointer, *defined_high*, rather than relying on presence bits or other synchronization of individual locations. We expand on each of these points below, and then remark on a property of the implementation that is shared by others in the literature, namely, its dependence on sequentially consistent shared memory.

The first point is that a *queue* may grow to hold an unbounded number of elements, because the *enqueue* operation dynamically allocates space as needed. Technically, the size is bounded by the total available memory on the machine, but this is a more generous bound than other published implementations. The implementation in [GLR83] uses a circular buffer, so there is a bound on how many elements may be stored in the *queue* at a given time, while the implementation in [HW90] is bounded in the total number of *enqueues* that may be performed. Dynamic object growth is particularly important in symbolic applications, where the size of data structures is difficult to predict in advance. Note that our implementation solves only the easier half of the dynamic object problem, since our implementations do not perform explicit deallocation.

The second point about our implementation is that it solves the problem of restricting access to filled slots without using presence bits or locking at the slot level, and instead, the *defined_high* pointer controls access. The problem arises, for example, when a *queue* is empty and both *enqueue* and *dequeue* are invoked concurrently: although *enqueue* may have been assigned a slot in the *queue*, the *dequeue* operation must wait for the slot to be filled. Herlihy and Wing's implementation depends on memory being cleared when allocated, because *dequeue* tests slots

to determine whether they are full or empty [HW90]. Our implementation runs correctly even if the allocator returns uncleared memory. We implemented a version that depended on cleared memory in place of the *defined_high* pointer, but found no measurable performance difference between the two implementations. This is in spite of having a memory allocator on the Firefly that automatically clears memory when allocated, so we were paying the price of initializing memory but not relying on it. In general, we prefer to avoid reliance on this aspect of the memory allocator since it incurs a overhead on all allocations.

A number of concurrent *queue* implementations are also described in [GLR83], each of them providing a different solution to the the problem of preventing unfilled slots from being *read*. All of the solutions involve an explicit flag (or some kind of counter) associated with each slot to denote whether the slot is empty or full. Thus, an initialization phase is required during which the buffer's flags or counters are set. In a bounded buffer *queue* this may be reasonable, but in a dynamically allocated one it will considerably slow the execution of *enqueue* operations whenever new buffer space must be allocated.

A final point about the *queue* implementation in Figure 3-7 is that it uses a strong notion of shared memory, since not all shared memory access are protected by critical regions. The *defined_high* pointer, which acts like a *counter*, is implemented as a simple *location* without locking. The process of incrementing *defined_high* need not be protected by a *lock* because each thread executing an *enqueue* waits for the value of *defined_high* to be one less than i, and then performs the increment. The index i is unique, since it was generated by the *counter*$*inc* operation on *high*, so the sequential consistency of memory guarantees that two increments will never be performed concurrently.

Each *enqueue* operation waits for all previous slots to be filled before returning, and thus our implementation is not wait-free, unlike the implementations that rely on presence bits or locks [HW90] and [GLR83]. In our implementation, if processors run at different rates, threads running on fast processors will be slowed to the pace of slower ones when accessing the *queue*. Page faults or remote memory accesses can, in effect, cause one processor to run slower than another, while wait-free implementations will continue to make progress in spite of these differences in execution speed. We give up the advantages of a wait-free implementation in return for fewer synchronization operations in one case [GLR83], and a less sophisticated (and poten-

dyn_array = **datatype** [t: **type**] **has** create, addh, fetch, size, print

interference between: (addh, print)

create = **procedure** () **returns** (dyn_array)
ensures: returns an empty array

addh = **procedure** (a: dyn_array, e: t)
ensures: adds e to the top of a

fetch = **procedure** (a: dyn_array, i: int) **returns** (t)
requires: $0 \leq i < \text{size}(a)$
ensures: return the i^{th} element of a

store = **procedure** (a: dyn_array, i: int, e: t)
requires: $0 \leq i < \text{size}(a)$
ensures: replace i^{th} element of a by e

size = **procedure** (a: dyn_array) **returns** (int)
ensures: returns the size of a

print = **procedure** (a: dyn_array)
ensures prints the elements of a in order

Figure 3-8: Specification of dynamic arrays (*dyn_arrays*).

tially more efficient) memory allocator in another case [HW90]. While the notion of a wait-free implementation is semantically quite powerful, the practical importance has not been proved. Wait-free implementations depend on synchronization primitives that are themselves wait-free, yet these do not exist on current architectures. For example, a wait-free implementation of *fetch-and-add* must allow for processors that crash while executing the *fetch-and-add*.

3.3.4 Arrays

A slight variation on the *queue* abstraction is an array that grows dynamically. The implementation of a *dyn_array* is a straightforward modification of the *queue* implementation, so we do not include it here. (The main difference is that the *low* pointer is not needed.) However, it is instructive to look at the specification of a *dyn_array*, which is shown in Figure 3-8.

The main difference between the *queue* and the *dyn_array* is that the *dyn_array* has *fetch* and *store* operations, but no operation for removing elements. Both *fetch* and *store* require that the index i is between 0 and $size(a) - 1$ to ensure that there is an element at i. The

precondition (*requires*) on a is unusual, since as noted ealier, it is difficult for the caller ensure a precondition on a shared mutable object. In this case the condition can be guaranteed by comparing the index against the size of the array before calling *fetch* or *store*. Because the size is nondecreasing, a precondition that holds at some instant in time will also hold in the future. Although they are not shown in the figure, safe versions of *fetch* and *store* that check whether i is in bounds are also implemented. In that case the monotonicity of an array's size is still being used, but only within the implementation. The cost of the safe operations is an extra comparison, which may be unnecessary if other information about the program execution can be used ensure the *requires* clause.

In general, objects that can only be mutated monotonically in a predictable direction are easier to use and reason about than objects that have arbitrary mutation patterns. A more familiar parallel programming technique that relies on a trivial kind of monotonicity is the use of objects that are initially undefined, but once they become defined they are never mutated [ANP87]. An even more degenerate case of monotonicity is an immutable object.

3.3.5 Mappings

A *mapping* type stores a set of domain/range pairs for arbitrary types of domain and range elements. There are interesting features of our *mapping* type in both its specification and its implementation. The implementation is discussed in detail below, but in discussing the specification we need to make one observation about the implementation: the representation of a *mapping* contains thread-specific data for each thread that uses the *mapping*. The use of thread-specific data has important implications for the specification; it changes the nature of the abstraction, since each thread must have its own version of the object. We refer to such objects as *multi-ported objects* and to each thread's version as a *port*. Implementations of this kind are prevalent in implementations of shared and distributed objects but we know of no other work that describes the specifications of the procedures that access these object. (See [Ell85, Her90, CD90, WW90] for some examples of objects that contain thread-specific data.)

The specification of a multi-ported object typically has an interference specification that depends on the port being used, which is important because it means that a multi-ported object looks different than a normal object, even at the abstract level. Consider the *mapping*

mapping = **datatype** [domaintype, rangetype: **type**] **has**
create, add_port, assign, print

interference between: (print, assign), (add_port, assign),
(add_port, add_port)
interference on ports between: (assign, assign)

create = **procedure** () **returns** (mapping)
ensures: returns a new identify mapping

add_port = **procedure** (m: mapping) **returns** (mapping)
ensures: returns a unique port to m

assign = **procedure** (m: mapping, d: domaintype, r: rangetype)
signals (bound(rangetype))
ensures: if d is bound to some r' in m
then signals bound(r')
otherwise the binding $d := r$ is added to m

print = **procedure** (m: mapping)
ensures: prints m

Figure 3-9: Specification of *mappings*

specification in Figure 3-9. All ports of a given *mapping* object access the same abstract value in that an *assign* to one port is observed by all others, but the interference specification distinguishes between different ports, since *assigns* interfere on the same port. Interference relations in previous examples were denoted by relations on procedure names, but a more expressive linguistic mechanism is needed for multi-ported objects. The intended meaning of the interference specification on *assigns* in Figure 3-9 is to permit executions in which there are concurrent *assigns*, but only when they are invoked on different ports. In practice, each thread using a particular multi-ported object is given its own port, so the sequential nature of threads will automatically ensure that the interference specification is observed. If there were a large number of threads accessing the object with low frequency, we could also establish a convention whereby the threads are partitioned by ports, and threads on the same port must go through a shared lock. In either case, there is more potential concurrency than if we had implemented the *assign* operation with a critical region protected by a single lock.

In addition to the interference specification on *assign* operations of the same port, there are also interferences of the kind we have discussed in other examples. Interference is specified

between *print* and *assign*, *add_port* and *assign*, and between *add_port* and *add_port*. These operations interfere regardless of the port being used.

An implementation of *mapping* is given in Figures 3-10 and 3-11. Code for the *print* procedure is omitted, since the interference specification on *print* and *assign* make a straightforward implementation possible. The representation of *mapping* type is given in Figure 3-10: each thread has its own record object, but they share all state except the *myself* index. The record contains an *announcement board*, which is an array of possibly null domain elements. There is one entry in the announcement board for each port, meaning one entry for every possible concurrent *assign* invocation; the *myself* index give the port's location for posting announcements.

In addition to the announcement board and associated index, the representation contains a dynamic array of *bindings*. *Bindings* that are stored in *elts* are part of the current *mapping*, while domain elements in *announce* are values that a thread is trying to bind, but for which the binding may not succeed.

The *assign* procedure is the most interesting algorithmically, and also the most important to the performance of the matching application in which *mappings* are used. The difficulty is in handling duplicate assignments correctly without forcing assignments to distinct domain elements to wait for one another. Two assignments are *duplicates* if they involve the same domain element, and two different instances of the internal *assign_help* procedure *collide* if their assignments are duplicates and they are concurrent. The *assign* procedure uses a backoff strategy to prevent duplicate assignments from being stored; the procedure repeatedly attempts the assignment by invoking *assign_help* and waiting between attempts. An *assign_help* operation has one of the following three behaviors:

- If *assign_help* collides with another *assign_help* operation, at least one of them signals *collision*.
- If the new *binding* is not a duplicate with anything in *elts*, *assign_help* adds the *binding* to *elts* and then returns.
- If the new *binding* is a duplicate with something in *elts*, then *assign_help* signals *duplicate*, passing the pre-existing range value back to the caller as part of the signal.

Assign_help is guaranteed to only add an assignment if the domain value hasn't already been

```
mapping = cluster [domaintype, rangetype: type] has
                  create, add_port, assign, print

    rep = record[myself: int,
                  announcements: array[domaintype],
                  elts: dyn_array[binding]

    binding = record[domain: domaintype,
                     range: rangetype]

    create = procedure () returns (mapping)
        m: rep := alloc(sizeof(rep))
        m.myself := 0
        m.announcements := array$addh(array$new(), null)
        m.elts := dyn_array[binding]$create()
        return(m)
      end create

    add_port = procedure (m1: cvt) returns (mapping)
        m2: rep := alloc(sizeof(rep))
        m2.myself := array$size(m1.announcements)   % get unique id for myself
        m2.announcements := m1.announcements        % share announcements
        m2.elts := m1.elts                          % share elts
        array$addh(m2.announcements, null)    % add place for my announcements
        return(m2)
      end add_port
```

Figure 3-10: Implementation of *mappings*—representation and creation

assigned, and isn't being assigned concurrently. If two invocations of *assign_help* collide, it is possible that one will succeed in making the assignment, or that neither will succeed, and instead, both will signal *collision*. In the latter case, both instances of *assign* that invoked *assign_help* will try again later. The delay between *assign_help* invocations increases with each failed attempt, and the delay is a function of the port, so the likelihood of collision decreases with each attempt.

Technically, the *assign* procedure is does not meet the *non-stopping* condition of Chapter 2, but given a reasonable model of the multiprocessor and its scheduler, an argument can be made that the probability of *assign* running forever approaches zero. In practice, we never observed an execution in which an assignment ran forever, even when *assigns* were done with artificially heavy loads. We experimented with a number of backoff schemes (denoted by the function f in Figure 3-11) for the delay, including linear, exponential, quadratic, and random, before settling on a quadratic scheme.

This implementation was chosen after experimenting with a much simpler algorithm that uses a single critical region for all *assigns*. Although in isolation the latency of a single *assign* operation is lower in the simpler scheme, the throughput is must worse. In the simple scheme each *assign* operation executes serially, even when the domain element was previously assigned so that no mutation is needed. In the more complicated implementation with backoff, non-mutating *assigns* can execute concurrently, with concurrency bounded by the number of ports. Even when two concurrent invocations must both mutate the *mapping*, if their two domain values are distinct, then the only time they must be serialized is during the *addh* operation, which we know from earlier discussion is a few instructions. The performance advantage of the more complicated strategy is noticeable not only under artificial loads, but also in the context of the matching application.

Another implementation that we considered but did not implement was to use a hash table and to lock at the level of buckets. This would probably be a reasonable choice, given the experiments that were performed with the matching program in Chapter 4. One reason we did not choose a hash table is that matching is not a stand-alone application, but occurs as a frequent operation in term rewriting and other symbolic applications. The *mappings* that occur in practice are quite small, while the key space is potentially large, and the number of

instances of substitutions is large, so the space requirements of a hash table might be excessive. In addition, assuming an imperfect hash function, two assignments on different domain elements may be serialized by the lock on buckets, and all bindings that are stored in that bucket will be compared while holding the lock. In contrast, our implementation only serializes during the *addh* operation or when two identical domain values are being assigned. An interesting future project would be to analyze the performance of the hash table implementation carefully, using the characteristics of real applications.

While these are interesting properties of the *mapping* implementation, we believe that the *mapping* type is most important to demonstrate the idea of multi-ported objects. In this case, the multi-ported nature is used to reduce the contention that occurs in the simpler implementation. Although we avoided the details of memory allocation in the code for *queues*, *dyn_arrays*, and *mappings*, a multi-ported memory pool is used to reduce contention there as well.

Multi-ported objects would also be useful as the abstract view of a distributed object. Distributed objects spread the implementation across nodes of a distributed system or distributed memory multiprocessor, using combinations of replication and data partitioning to make the distributed objects behave as if it were a single shared object. The most common example of a distributed object is a memory cell in a multiprocessor with caches, but many others have been designed and implemented for both distributed systems [Ell85, BT88, BHJ+87] and for multiprocessors [Luc87b, Dal86, CD90]. To access a distributed object, these systems provide sophisticated run-time support so that each object is given a single name; the run-time system must determine what node in the distributed object should be used for a particular operation. This determination can be done using various heuristics to avoid overloading nodes and to achieve high locality. Instead of requiring the overhead of such run-time support, multi-ported objects suggest a much simpler system, and slightly more complicated programming model, in which each thread has information about where to locate its node (i.e., port) of the distributed implementation. Thus, multi-ported objects may be useful for improving locality on distributed memory machines, just as they reduce contention on shared memory machines.

```
assign = procedure (m: mapping, d: domaintype, r: rangetype)
            signals (bound(rangetype))
        backoff: int := m.myself
        while true
            {for (i := 0) to (array_concurrent$size(m.elts) - 1)
                {b: binding := m.elts[i]
                if b.domain = d then signal bound(b.range)}
            assign_help(m, d, r)
                except when duplicate(r2): signal bound(r2)
                except when collision:
                    {wait(backoff)          % pause for backoff clock ticks
                    backoff := f(backoff)
                    continue}
            return}
         end assign

assign_help(m: mapping, d: domaintype, r: rangetype)
            signals (collision, duplicate(rangetype))
        {m.announce[m.myself] := d
        for (i := 0) to (array$size(m.announce) - 1)
            {if i = m.myself then continue   % skip own announcement
            d2: domaintype := m.announce[i]
            if d2 = null then continue
            if d = d2 then    % found collision
                {m.announce[m.myself] := null signal collision}}
        for (i := 0) to (array_concurrent$size(m.elts) - 1)
            {b: binding := m.elts[i]
            if (b.domain = d) then   % found stored duplicate
                {m.announce[m.myself] := null
                    signal duplicate(b.range)}}
        array_concurrent$addh(m.elts, binding$[domain: d, range: r])
        m.announce[m.myself] := null
        return
         end assign_help
```

Figure 3-11: Implementation of *mappings*–the *assign* procedure

Chapter 4

Designing Parallel Programs

In the previous two chapters we considered the question of what basic building blocks should be used for parallel programming, and chose the notion of linearizable concurrent objects. In this chapter we consider the problem of parallel program synthesis, or how to design programs that have the modularity provided by concurrent objects, but without sacrificing performance. Several other approaches to parallel programming have been proposed, and each one appears to be suited to some restricted domain of applications. Outside their intended domain, programs produced using these approaches tend to be either inefficient or overly complex. Our approach is no exception; it is designed programs with irregular patterns of control and communication, and in particular, for symbolic programs.

As discussed in Chapter 1, the trade-off between program simplicity and program performance is even more significant for parallel programs than for sequential ones. One way of comparing various programming approaches is to consider how that trade-off is resolved. Our approach is rather liberal in terms of programmer freedom, because it is based on a model of asynchronous, explicit parallelism in an imperative programming language. Thus, in comparison to approaches based on safer, more restricted programming models, such as a purely functional language or a strict data parallel model, we have given up a certain amount of simplicity in the belief that better performance can be achieved.

We address the problem of program simplicity, which is partially a concern about correctness, by breaking the design process into distinct stages. Each stage has a clearly defined set of requirements to be met, and the argument that a particular design meets these requirements

can be made either formally or informally. The resulting programs have a stylized module structure that makes them easier to modify, debug, and maintain.

In Section 4.1 we describe a class of programs that is well-suited to our approach. In Section 4.2 we present an overview of the *transition-based approach*, and in Section 4.3 we apply it to a substantial example. A discussion of the approach follows in Section 4.4.

4.1 Program Characteristics

Roughly speaking, parallelism can be divided into *process parallelism*, which involves executing different processes in parallel, or *data parallelism*, which involves executing identical processes on different data in parallel. Process parallelism has limited scalability, since different code must be written to implement each process; a safe conclusion is that programs involving thousand-fold parallelism will not be implemented solely with process parallelism. It is a mistake, however, to conclude that process parallelism is uninteresting. In many applications the parallelism with the coarsest granularity, and thus the lowest relative overhead, is process parallelism.

Process parallelism includes the software pipelining paradigm, wherein the computation is divided into a fixed set of stages, the processes at the stages execute in parallel, and data is passed from one stage to the next to be processed. Consider, for example, a sequential program of the form $P_1; P_2; P_3$, where each of the P_i is a procedure invocation. In addition, assume that each of the P_i's involves a startup phase, during which data is being dispersed to processors, and an expiration phase, during which some processors are waiting for others to finish. The parallelism profiles for the P_i's have tails of poor processor utilization at both ends of the computations. If we parallelize the program by parallelizing each P_i, but leave the sequential structure at the highest level, then the parallelism profile of the program will be the concatenation of the P_i profiles, and any periods of low processor utilization will be inherited from the P_i's. If, instead, parallelism is added to the program level by overlapping the P_i's, then the periods of low utilization in one P_i may be masked, effectively smoothing the program's parallelism profile.

For applications involving irregular computations, the impact of poor processor utilization is higher than with regular computations. The reason is that irregularities make the scheduling problem more difficult, and therefore lead to periods of low processor utilization. The large

data structures in numerical programs are usually arrays of numbers, for which the cost of operations can be predicated in advance. When the cost of an operation is independent of its input, or at least bounded for any input, a static scheduling strategy may be effective. In contrast, the large data structures in symbolic programs are often trees or graphs, and since the size and layout of these structures is not known at compile time, scheduling must either be done dynamically or badly. Data parallelism is therefore easier to schedule in numerical programs, where the size of the input is constant for each element in an array and grain size can be adjusted uniformly by working on fixed-size sub-arrays. There is a second phenomenon that occurs in symbolic problems, in that performance varies significantly even for inputs of the same size. Consider the problem of checking for tree isomorphism; if the trees differ at the top, then the operation will be fast, even for large trees. This kind of *performance instability* occurs in most search problems and problems for which some inputs result in abnormal termination; while such problems are typical of symbolic computations, they arise in some numerical ones as well.

Our approach, introduced in Section 4.2, incorporates both data and process parallelism within a single uniform framework, and is intended for development of coarse grained parallel programs. One of the challenges was to make effective use of processor resources, even when programs involve heterogeneous data structures and operations that exhibit performance instability. The approach is not intended for applications having easy data parallel implementations, where the bulk of the computation is a set of independent and uniformly expensive operations. Although there is nothing preventing its use on such problems, little or no benefit will be derived from the generality of the approach. We believe that programs such as compilers, theorem provers, algebraic manipulation systems, circuit simulators, and expert systems may all be amenable to our approach.

We make one simplifying assumption about our application domain: the programs are not reactive. The programs run in a batch style, and do not accept input from the environment or from a user after the initial input values. This will simplify both the presentation and the performance analysis of resulting programs, but as discussed in Section 4.4, the restriction is not fundamental to our approach.

4.2 A Transition-Based Approach

In this section we present an overview of our approach to developing parallel programs. The presentation is informal and only intended to give the reader the intuition behind the approach. A more thorough description is given through an example in Section 4.3. A larger example is given in Chapter 5.

In the previous section we identified the problem of producing programs that exhibit coarse grained parallelism and make effective use of available processor resources. The purpose of transition-based development is two-fold: to reveal coarse grained parallelism to the program designer, and to provide a framework for clean, modular implementations.

A common approach to parallel program design is to take a sequential program and add parallelism and synchronization such that the parallel program is, in an abstract sense, equivalent to the sequential one. (Note that parallel algorithms, in contrast to parallel programs, are often designed from scratch.) The result of this development technique is that a parallel program tends to retain a sequential structure at the highest level, in effect inheriting synchronization points from the sequential implementation. Since sequential programming languages require that programmers order all operations, even when the order is arbitrary, these synchronization points may unnecessarily limit parallel program performance.

In our approach the parallel program designer also starts from a sequential program and adds parallelism and synchronization, while preserving an equivalence to the original. The difference is that we start with a highly nondeterministic, abstract description of a sequential program, without the arbitrary orderings of a conventional sequential program. These abstract programs are called *transition axiom specifications*, and our approach is called the *transition-based approach*. The four basic steps in the approach are outlined here:

1. Give a *transition-axiom specification* for the problem.

2. Refine the specification.

3. For each transition axiom in step 2, implement a procedure that effects the specified state change.

4. Implement a scheduler to execute the procedures in step 3.

A transition-axiom specification describes the set of possible state transitions that will be observed in any execution. It is an abstraction of a program, rather than a specification of a problem. For example, a specification of a program that sorts an array, A, might state that any pair of elements, $(A[i], A[j])$, can be swapped if $i < j$ and $A[i] > A[j]$. Transitions are specified using *transition axioms*, which are guarded commands [Dij76]. In our use the commands are flat, that is, there is no nesting of guards. For example, a specification of the sorting program contains a single axiom, written $(i < j)\&(A[i] > A[j]) \rightarrow (A[i] := A[j])\&(A[j] := A[i])$. A transition-axiom specification also constrains executions with liveness properties, which cannot be expressed as transition axioms. A liveness property requires that certain transitions eventually happen. For example, in any execution of the sorting program, a pair of elements cannot remain forever out of order; eventually, either they must be swapped with each other, or one of them must be swapped with another element. The specification in step 1 is a description of the high-level actions in an execution, and the difficulty of writing this specification should be less than that of writing a traditional sequential program to solve the same problem.

Transition axioms specifications are used commonly to describe systems with concurrency, e.g., [Lam89, Lam90, LF81, LT87, CM88]. Our transition axiom specifications do not differ significantly from any of these. However, we allow a general class of liveness properties as in Lamport's language [Lam90], rather than the fixed notion of weak fairness in Unity [CM88] and the I/O Automata model [LT87]. Moreover, our specifications require a richer semantic model than provided by the formal languages of [CM88] and [Lam90], because, for example, we allow sharing among objects in the state [GL90]. The emphasis in most of these methods is on reasoning, rather than development, and on algorithms, rather than large programs. Our emphasis is on development of large programs, with concerns for both modularity and efficiency.[1]

In step 2, the transition-axiom specification is refined, possibly a number of times, with the goal of finding a specification that can has a clean and efficient *direct implementation.* The goals of refinement are: simple guards, an appropriate granularity of transition axioms, and a weak liveness property. We postpone making these goals precise or giving examples until after

[1]Chandy and Misra address performance concerns in Unity [CM88]; the differences between their approach and ours, with regard to performance, appear in the later steps of our approach.

presenting steps 3 and 4, since the goals of specification refinement are tied to the question of what it means for a program to implement a specification. Given the relationship between specification and implementation, we show that some transition-axiom specifications lead to implementations with either poor performance or ugly abstractions. It is such specifications that the designer tries to avoid in step 2; when intuition fails, it may be necessary to repeat step 2 if the program written in steps 3 and 4 is unsatisfactory.

In a general sense, the purpose of refinement in step 2 of our approach is the same as the purpose of refinement in Unity program development: in both cases the goal is to find a specification that is closer to a real implementation, and in both cases the designer uses an understanding of the process by which specifications are translated to implementations in determining how a specification should be refined. There are two significant differences between the approaches. The first difference is that with Unity, the translation process targets a particular machine architecture, while in our approach, the translation targets an abstract state machine that is implemented by a programmer. Thus, refinement in our method stops at a higher level of abstraction; it stops as soon as the programmer is confident that an abstract state machine with the necessary operations can be efficiently implemented. The second difference is that in Unity, translations are viewed as automatic; the intent, which has not yet been realized, is for Unity compilers to produce executable multiprocessor programs from Unity programs. Thus, the translation process in Unity is necessarily conservative; it must be correct for any Unity program, and it must be decidable. In our approach, the translation between a specification and implementation is constrained only by the correctness conditions on the implementation, and by the programmer's ingenuity, since the programmer does the translation process by hand.

Between step 2 and step 3, we depart from specifications and move on to programs. Although the specifications describe executions, they are not executable. The transition axioms define what states may follow others, but for programs more complicated than a sorting procedure, there may be non-trivial computation in getting from one state to the next that is not described in the specification. Furthermore, the liveness properties describe properties of infinite executions, not procedures for ensuring those properties. It is popular in the literature to refer to lower-level specifications as implementations of higher-level ones [Lam89, CM88, LT89],

but we reserve the word implementation for programs written in a language that runs on a real machine.

In step 3, a set of procedures called *transition procedures* are implemented. The procedures are themselves sequential, but are run concurrently with one another. There is one procedure for each transition axiom. If the guard is true, the state is changed to match the given state transition, and if the guard is false, the state remains unchanged, and an exception is signaled. For the swap axiom, the corresponding procedure would take the array and two indices as input, compare the array elements at those indices, and swap them if they are out of order. Since two concurrent instances of this procedure could interfere with one another, some synchronization would be needed. In addition to the set of transition procedures, there is a *termination* procedure that returns true if all the guards are false, and returns false if any one of them is true.

The scheduler implemented in step 4 is a multi-threaded program that invokes instances of the transition procedures. Thus, the programming model involves a single parallel module (the scheduler) invoking operations (the transition procedures) on a concurrent object (the program state). Application-specific scheduling has been demonstrated to result in better performance than when systems schedulers are used directly [And89], and in particular, the *worker* model of application scheduling has been used successfully in a number of parallel applications [MNS87, CG89, RV89, JW90]. The worker model is an example of a user-level scheduling paradigm; it's definitive characteristic is that the tasks being scheduled are short-lived, so each is allowed run until it either blocks or finishes. Thus, the worker model avoids the complication and overhead of time-slicing schedulers. We take the simplicity of the worker model one step further by requiring that transition procedures are non-waiting, thereby eliminating the need for blocking.

A key feature of our approach is that a weak correctness requirement is placed on the scheduler, so that there is a great deal of freedom in the choice of a scheduler. This allows for significant performance tuning of the scheduler, which occurs late in the design process. Our schedulers are implemented so that any transition procedure with a true guard will eventually be executed, and if there are no transition procedures with true guards, then all scheduler threads will eventually halt. No other constraints are placed on the scheduler: it may invoke any transition procedures concurrently without causing race conditions or deadlock. Each scheduler

thread interleaves transition procedure invocations with the *termination* procedure, and halts when *termination* returns true. A naive scheduler for a sorting program might repeatedly cycle through all pairs of indices, calling a swap procedure for each pair, until all invocations fail. A more efficient scheduler would keep a history of unsuccessful swaps, so that it could avoid retrying pairs of indices that failed in the past and have not been affected by other swaps since that time.

4.3 Design of a Term Matching Program

To make this discussion concrete, we demonstrate the transition-based approach on a program for term matching. The matching program is useful for pedagogical reasons: it is a simple problem, but characteristic of symbolic programs, and the implementation is large enough that modularity issues can be illustrated. While the parallel implementation achieves good performance for very large inputs, there is no advantage to parallelism for inputs of a more realistic size. Each of Sections 4.3.1–4.3.4 describes one of the four steps in the program's development. In general, there are non-trivial arguments to be made for the correctness of each step in the approach; Section 4.3.5 illustrates some of these. First, though, we briefly introduce the matching problem.

Matching procedures take terms as inputs and produce substitutions as output. Terms are built from a fixed set of variables, V, and function symbols, F, where each function symbol has an associated arity. A *term* is either a variable, or it has the form $f(t_1, ..., t_n)$, where f is a function symbol of arity n (written $arity(f) = n$), and each of the t_i is itself a term. Function symbols having arity zero are *constants*, and are written as terms without parentheses. It is convenient to think of terms as trees, and for a given term t we refer to the function symbol at the root as the *head*, and subtrees immediately below the root as the *child subterms*. Given a term t the i^{th} child subterm is denoted $t[i]$, the head symbol is denoted $t.head$, and when $t.head$ is a function symbol, $arity(t.head)$ is simply written $arity(t)$. A term that contains no variables is said to be *ground*. Throughout this chapter, x represents a variable, a and b are constants, and p and t are terms; all may appear with subscripts. We use the predicates $is\_var(t)$ and $is\_ground(t)$ with the obvious meanings.

A *substitution* is a mapping from variables to terms, denoted by a set of variable/term

pairs. For example, $\{x_1/t_1, ..., x_n/t_n\}$ is the substitution that maps each x_i to t_i, and all other variables to themselves. Substitutions are extended to a mapping from terms to terms as follows. For a substitution, σ, the application of σ to t is written $\sigma(t)$ and by definition, $\sigma(f(t_1, ..., t_n)) = f(\sigma(t_1), ..., \sigma(t_n))$. The *domain* of a substitution σ is the set of variables that are changed by σ, $\{v \mid \sigma(v) \neq v\}$. The composition of substitutions s_1 and s_2, denoted $s_1 \circ s_2$, is defined by the equation $(s_1 \circ s_2)(t) = s_2(s_1(t))$.

The matching problem is to take a term, p, called the *pattern*, and a ground term t, called the *target*, and determine whether there exists a substitution σ such that $\sigma(p) = t$. In most applications, the value of the matching substitution is needed. Therefore, a *matching procedure* is specified to take a pattern and target as input, and either produce a matching substitution, or raise the exception *no_match*. We constrain the matching substitution so that its domain is a subset of the variables in the pattern, thereby making the matching substitution unique.

For example, given a pattern $f(g(a, x), x)$, and a target $f(g(a, b), b)$, where x is a variable and a and b are constants, the matching substitution is $\{x/b\}$. Using the same pattern, but a target term $f(f(a, b), b)$, there is no match because of a clash between the inner f and g. If, instead, the target term is $f(g(a, b), a)$, there is again no match; in this case the failure is caused by the inconsistency of having x mapped to both a and b.

4.3.1 A Transition-Axiom Specification for Matching

In this section we apply the first step in the transition-based approach by giving a transition-axiom specification of a matching procedure. A matching procedure, whether sequential or parallel, performs the same basic actions. Consider the sequential matching procedure shown in Figure 4-1. Rather than returning a substitution, *match* is called with the empty substitution and modifies it to be the matching substitution, when one exists.

There are three actions performed in *match*: function symbols in p and t are compared (line 7); variables in p are bound to a subterm of t and added to s (line 3); and for any variables that occur more than once, their corresponding subterms are checked for equality (line 5). The *ground_equal* procedure checks two ground terms for equality and is included as part of the problem to be parallelized. *Ground_equal* has the action of comparing function symbols in the two inputs (line 12).

```
match = procedure (p, t : term, s : substitution) signals (no_match)                1
        if is_var(p)                                                                 2
            then assign(s, p, t)                                                     3
                 except when already_bound(t2) :                                     4
                      if ground_equal(t, t2) then return                             5
                           else signal no_match                                      6
        if (head(p) ≠ head(t)) then signal (no_match)                                7
        for (i := 1) to arity(p)                                                     8
            match(p[i], t[i], s) resignal no_match                                   9
    end match                                                                        10

ground_equal = procedure (t1, t2 : term) returns (bool)                              11
        if (head(p) ≠ head(t)) then return(false)                                    12
        for (i := 1) to arity(p)                                                     13
                 if ¬ground_equal(p[i], t[i]) then return(false)                     14
        return(true)                                                                 15
    end ground_equal                                                                 16
```

Figure 4-1: A Sequential matching procedure

The call to *assign* in *match* demonstrates the use of signals. For convenience, we have included the interface specification for *assign*, which was presented in Chapter 3, in Figure 4-2. When the *already_bound* signal is raised by *assign*, it returns a value of type *term*. The clause in *match* that begins "*except when already_bound*($t2$)" catches the *already_bound* signal, assigns the signaled term value to the local variable $t2$, and passes control to the statements nested within the clause. The *resignal* statement in line 9 is shorthand for catching and then signaling the *no_match* signal.

```
assign = procedure (s : substitution, v, t : term) signals (already_bound(term))
        requires: is_var(v) & is_ground(t)
        ensures: if v ∈ domain(s)
                      then signals already_bound(s(v))
                      else s := (s ∘ {v/t})
```

Figure 4-2: Interface of *assign* procedure for substitutions

The need to do assignment in matching makes a parallel solution non-trivial; a pattern that contains multiple instances of the same variable may result in simultaneous assignments to that variable.

In writing a transition-axiom specification for the matching problem, we can use the same actions as in the sequential *match* procedure. A *transition-axiom specification* contains a list of state components (typed object names), a predicate giving initial values for the objects, a set of flat guarded commands, and a liveness property. Figure 4-3 gives an example specification for matching, in which the problem is viewed as a transformation on a *container* of equations [CL88], and each transformation is one of the actions from the sequential code. (The symbol $\doteq$ is used as part of an equation literal, to avoid confusion with expressions that contain equality tests.) The *container* starts with the single equation to be solved, and when a match exists, it ends with a representation of the matching substitution. A matching substitution is represented by a *container* in *solved form*: a *container* in which all equations have a variable on the left-hand side and a ground term on the right-hand side, and no variable occurs more than once.

We give E the type *container*, rather than *queue*, *stack*, *set*, or *bag*, for example, because *containers* are less constraining. *Containers* were specified in Chapter 2 (Figure 2-1): they have an *insert* operation and a *choose* operation. Unlike *queues* or *stacks*, the order in which elements are removed from a *container* is not constrained by its specification; by leaving this order unconstrained, we are postponing scheduling decisions. Unlike *sets* or *bags*, *containers* do not treat duplicate elements specially; by treating all equation objects as distinct elements, we are postponing the decision of whether special treatment *should* be given to duplicates, and if so, what the right notion of duplicate is. (In particular, two equations may be distinct even though they represent the same pair of terms.)

The first three parts of a transition-axiom specification (everything except the liveness property) define a nondeterministic state machine. A *state* is a well-typed assignment of values to the state components, and an *execution* of a state machine is a possibly infinite sequence of states. In Figure 4-3 the machine has a single state component, E, that can take on values of type *container*[*equation*] or a special value *no_match*.

The initial condition constrains the possible initial states of a machine. In Figure 4-3, the machine starts with E holding the single equation (*pattern* $\doteq$ *target*), which is the user's input pattern and target.

The transition axioms define the legal steps, or actions, of a machine. They are written as guarded commands, $G \Rightarrow S$, where G is the guard and S is the statement part of the command.

State Components

E : container[equation] + no_match

Initially

E = (pattern $\doteq$ target)

Transition Axioms

decomposition

$(p \doteq t) \in \mathrm{E} \Rightarrow$

if ($\neg$is_var(p) & (p.head = t.head))

then (E := E + ($p[1] \doteq t[1]$) + ... + ($p[n] \doteq t[n]$) − ($p \doteq t$))

clash

$(p \doteq t) \in \mathrm{E} \Rightarrow$

if ($\neg$is_var(p) & (p.head $\neq$ t.head))

then (E := no_match)

consistency

$(p_1 \doteq t_1) \in \mathrm{E}$ & $(p_2 \doteq t_2) \in \mathrm{E}$ & $\neg$same_obj$((p_1 \doteq t_1), (p_2 \doteq t_2)) \Rightarrow$

if (is_var(p_1) & is_var(p_2) & ($p_1 = p_2$))

then (**if** ground_equal(t_1, t_2)

then E := E − ($p_1 \doteq t_1$)

else E := no_match)

Liveness

Executions eventually terminate with E in solved form or equal to no_match

Figure 4-3: Transition-axiom specification for matching.

Semantically, an *action* is a pair of states, *pre* and *post*. An action is an *instance* of an axiom, $G \Rightarrow S$, if G is true in the pre state, and the post state can be derived from the pre state by "executing" S. An axiom is said to be *enabled* in a particular state if its guard is true in that state. Given a specification, an action is *legal* if it is an instance of some axiom, and an execution is *legal* if it starts with an initial state, and each consecutive pair of states is a legal action. Thus, state machines execute nondeterministically—at any point in an execution, any legal action may be taken. For convenience we give names to transition axioms, and refer to an instance of an axiom named A as an A action.

Consider the transition axioms in Figure 4-3. The *decomposition* and *clash* axioms are enabled if E is non-empty, and the *consistency* axiom is enabled if E contains at least two equations. The *same_obj* predicate in the guard of the *consistency* axiom is true only if the

two named equations refer to the same object. Two equations with the same syntactic terms are not necessarily the same object. If E is equal to *no_match*, we treat this as an empty set, so all of the guards are false.

If there is an equation in E for which the right and left-hand sides have the same head, then a *decomposition* action replaces the equation with the equations made up of the pairs of child subterms. A *clash* action detects function symbols that are not equal, and records this by changing the value of E to *no_match*. A *consistency* action is used to make E a well-defined substitution. If there is more than one equation involving some variable x, then the two terms made up of the right-hand sides of the equations are tested for equality. If they are equal, the two equations are redundant, so one of them is deleted; if they are not equal, an inconsistency has been discovered, so E is set to *no_match*. The statement parts of all three axioms contain conditionals without *else* clauses; when the given condition is false, the post state is equal to the pre state. Thus, in addition to the actions just described, all three axioms have stuttering actions as instances, where a *stuttering* action is one in which the pre and post states are equal [AL88].

Transition axioms, and the state machines they define, are useful as specifications, but they cannot express every interesting property of programs. Transition axioms define *safety* properties, which make a statement about what bad things cannot happen in an execution, e.g., what states cannot be reached. They cannot express properties such as termination or fairness, which are examples of *liveness* properties. A liveness property makes a statement about what good things must eventually happen in an execution. Given a (correct) specification, no legal execution will compute an incorrect answer, and any partial legal execution can be extended to one that computes the correct answer. To extend this partial correctness condition to total correctness, the liveness property must be taken into account. A legal execution for which the liveness property holds is said to be *live*.

The liveness property in Figure 4-3 restricts executions to those that terminate with E representing a solution. This property is the conjunction of two separate liveness properties: executions must be finite, and executions must have a state in which E is either in solved form or equal to *no_match*. The set of executions that are legal but not live contains executions that have an infinite sequence of stuttering actions, and those that end before E represents a

solution.

In general, it may not be obvious that a given transition-axiom specification specifies the desired program (is *correct*), or that it specifies any program at all (is *consistent*). The key to a partial correctness argument for specification 4-3 is the following invariant: considering E as a system of equations to be solved, the solution to the system of equations is the same in every state. Given partial correctness, total correctness is obvious, since termination is explicitly stated in the liveness property. A consistency argument for a specification involves showing that the set of live executions is nonempty. A consistency argument for specification 4-3 relies on the existence of a solved form representation for any matching substitution, and there being a sequence of actions by which the matching substitution or *no_match* can be computed.

4.3.2 Refining the Matching Specification

The purpose of the specification in step 1 is to capture the basic actions of the program, based on the program designer's intuition. In step 2, the purpose is to find a specification that retains those basic actions, but that more directly describes the structure of a proposed implementation.

Recall from the overview of our approach that we require a one-to-one mapping between the transition axioms in the specification of step 2 and the transition procedures of step 3. In Section 4.3.4 we define the notion of *direct implementation* that makes the mapping from specification to implementation more precise. This direct relationship between syntactic components of a specification and modules of an implementation is unique to our approach, and will give our programs a kind of modularity that would not otherwise exist. It will also raise some issues of implementability that do not arise when a more general relationship between specification and implementation is allowed.

Certain qualities of a specification will make it easier to find an efficient direct implementation: a weak liveness property, simple transition axioms, and an appropriate grain size for the target machine.

1. A weak liveness property allows for flexibility in scheduling, making it possible to do performance tuning of the scheduler late in the program development process. In matching we use a notion of *universal weak fairness*, which requires only that a state machine continue taking steps as long as it has legal steps to take. Lamport describes this notion

(without naming it) as "the weakest liveness property that appears in practice" [Lam90].

2. Simple transition axioms allow for efficient implementations of the transition procedures. In particular, the transition axioms guards should be quickly computable. Because our programs use the worker model, guards are typically predicates on the existence of work to be done, i.e., the existence of an element in a *container*. In addition, our guards contain predicates on scalar variables.

3. Choosing an appropriate grain size for the transition axioms is also a performance issue, because the parallelism in our programs comes from executing multiple instances of transition procedures in parallel, not from parallelizing within the transition procedures. The computation required by a given transition procedure is determined by its transition axiom, since the transition axioms define the possible post states that must be computed from a given pre state. If the computation specified by a single axiom is too high, relative to the total computation in the program, then the program will have insufficient parallelism. If the computation specified by a single axiom is too low, then the overhead of scheduling (which is a factor even for application-specific scheduling) may outweight the performance gains from parallelism.

A specification with all three of these subjective qualities is said to be *directly implementable*; as with many design methods, there is no precise criterion to say when a design admits a good implementation. In the remainder of this section, we describe some the techniques for obtaining a directly implementable specification for the matching application. The techniques are generally useful, and in particular, will be used again in the completion example of Chapter 5.

Weakening the Liveness Property

To motivate the need for a more refined specification of the matching program, we begin by considering a direct implementation of specification 4-3, and show that the specification has some undesirable features. The problem with the specification is that the axioms allow stuttering actions, and as a result, a strong liveness property is necessary to rule out executions with infinite stuttering actions. The liveness properties are implemented by the scheduler, and in general, having a strong liveness property may require an unimplementable, complicated,

or inefficient scheduler. For the specification in Figure 4-3, there are implementations of the transition procedures such that no scheduler exists. In the *decomposition* axioms, for example, the equation $(p \doteq t)$ can be any equation in the container—the choice of which equation is unconstrained. There is nothing to prevent an implementation of *decomposition* that repeatedly chooses the same equation every time it is called. A similar argument can be made for the *clash* and *consistency* axioms, so there are executions that interleave instances of all three axioms, and still make no progress. Given such transition procedures, it is impossible to implement a scheduler that ensures the liveness property. In a direct implementation, the scheduler only controls the order in which procedures are invoked; it cannot control nondeterministic choices made within the transition procedures.

To allow for flexibility in scheduling, a directly implementable specification should have a weak liveness property. In describing schedulers for matching, we assume the implemented specification has a particular liveness property called *universal weak fairness*. Universal weak fairness asserts that an implementation must continue to take actions as long as possible; as a property on executions, this means that any finite execution must end with a state in which no axioms are enabled. All other fairness notions of practical interest are stronger than universal weak fairness. In Lamport's Temporal Logic of Actions, universal weak fairness is denoted $\mathrm{WF}(\mathcal{N}_\Pi)$ for program Π; Unity and I/O Automata both have the stronger notion of *weak fairness*, which requires fairness between different actions. Stronger liveness properties such as weak fairness can, and in the completion example will, be used in directly implementable specifications. However, using universal weak fairness allows maximum flexibility in implementing the scheduler, and therefore also allows maximum performance tuning of the scheduler. Furthermore, if implementable specifications for different applications have the same liveness property, the ideas used in implementing one scheduler, and perhaps the scheduler itself, can be reused in other applications.

Specification 4-4 *satisfies* specification 4-3, in the sense that given the same pair of input terms, both specifications allow (only) finite executions, and the matching substitution produced by any execution of either specification will be the same substitution. A precise definition of satisfaction requires identification of the external states as in [AL88] or external actions as in [LT87], where the external states or actions are things observable by the user of the

State Components

E : container[equation] + no_match

Initially

E = (pattern $\doteq$ target)

Transition Axioms

decomposition

$((p \doteq t) \in \mathrm{E})$ & $(\neg \mathrm{is\_var}(p))$ & $(p.\mathrm{head} = t.\mathrm{head})$ $\Rightarrow$

$\mathrm{E} := \mathrm{E} + (p[1] \doteq t[1]) + ... + (p[n] \doteq t[n]) - (p \doteq t)$

clash

$((p \doteq t) \in \mathrm{E})$ & $(\neg \mathrm{is\_var}(p))$ & $(p.\mathrm{head} \neq t.\mathrm{head})$ $\Rightarrow$

$\mathrm{E} := \mathrm{no\_match}$

consistency

$((x \doteq t_1) \in \mathrm{E})$ & $((x \doteq t_2) \in \mathrm{E})$ & $\neg \mathrm{same\_obj}((x \doteq t_1), (x \doteq t_2))$ & $\mathrm{is\_var}(x) \Rightarrow$

if $\mathrm{ground\_equal}(t_1, t_2)$ **then** $\mathrm{E} := \mathrm{E} - (x \doteq t_1)$ **else** $\mathrm{E} := \mathrm{no\_match}$

Liveness

Universal weak fairness

Figure 4-4: Transition-axiom specification with weak liveness property

program. Once external states (or actions) are identified as such, a notion of satisfaction can be defined to require that those external states (or actions) be preserved. Because matching is a batch program, the only states of interest to the user are the initial and final states. Notions of satisfaction between transition axiom specifications have been extensively studied in the literature; the notions have been formalized in various models, and techniques for proving satisfaction, based on the idea of a *refinement mapping* have been described [Lam80, AL88, LF81, LT87, CM88]. In Section 4.3.5, we discuss satisfaction in more detail and give correctness arguments using refinement mappings [AL88]. In this section, we include only those ideas that are relevant to understanding what the various specifications mean, and how they differ.

Specification 4-4 has universal weak fairness as its liveness property, which is sufficient, in part, because stuttering actions are not legal. The predicates that appeared in conditional expressions in specification 4-3 now appear in the guards. (In specification 4-4, it is again the case that none of the axioms are enabled when E is equal to *no_match*, and the two equations named in the *consistency* guard refer to different equations.)

A state machine without legal stuttering actions does not necessarily make progress towards

computing the answer; it may, for example, have cycles in an execution even though any two consecutive states are distinct. Showing that universal weak fairness is sufficient for specification 4-4 is part of the correctness argument given in Section 4.3.5. Roughly, the argument is that each action makes the number of symbols in E smaller, so they will eventually "consume" all the equations that satisfy their guards. Furthermore, there are no infinite executions of any single kind of action, or of any subset of the actions. *Consistency* actions, for example, cannot be starved, since any execution made up only of *decomposition* and *clash* actions will eventually run out of equations with non-variable left-hand sides. For this reason, fairness between axioms (i.e., weak fairness) is implied by universal weak fairness; by stating the weaker notion of universal weak fairness, it is clear that the scheduler in step 4 need not interleave the different transition procedures.

In the final state of an execution from specification 4-4, when all the guards are false, E is in solved form or equal to *no_match*. Thus, since such a final state will always be reached, specification 4-4 satisfies the liveness property of specification 4.3.5.

Simplifying the Guards

We argued earlier that the liveness property in specification 4-3 made that specification inappropriate for implementation purposes, because a scheduler could not be implemented. In specification 4-4, the liveness problem has been eliminated, but a new problem has been introduced: it is unlikely that the transition procedures have clean and efficient implementations. We cannot give firm evidence on the point, but we suggest procedure implementations that illustrate some of the difficulties. An implementation of the *decomposition* axiom chooses and deletes an equation from the container and adds its child equations; it is not free to choose any equation, but only those having equal head symbols on both sides. Similarly, an implementation of the *clash* axiom must choose only equations in which the head symbols are different. Most of the complexity of the matching problem has been pushed into the data structures; it is unclear that we are closer to solving parallel matching than when we started.

The problem with specification 4-4 is that the guards refer to properties of the elements within the container, rather than properties of the container alone. This makes it efficient implementations difficult. Specification 4-5 avoids this problem, since the guards depend on

State Components

E : container[equation] % equations
B : container[equation] % bindings
S : container[equation] % substitution
C : {running, no_match} % clash flag

Initially

E = (pattern $\doteq$ target)
B = S = $\emptyset$
C = running

Transition Axioms

declash
$((p \doteq t) \in \mathrm{E})$ & $(\mathrm{C} \neq \mathrm{no\_match}) \Rightarrow$
if (is_var(p)) **then** $(\mathrm{E} := \mathrm{E} - (p \doteq t)$ & $\mathrm{B} := \mathrm{B} + (p \doteq t))$
else if (p.head = t.head)
then $\mathrm{E} := \mathrm{E} + (p[1] \doteq t[1]) + ... + (p[n] \doteq t[n]) - (p \doteq t)$
else C := no_match

consistency
$((x \doteq t_1) \in \mathrm{B})$ & $(\mathrm{C} \neq \mathrm{no\_match}) \Rightarrow$
if $(\exists\ t_2 \mid (x \doteq t_2) \in \mathrm{S})$ **then**
(**if** ground_equal(t_1, t_2) **then** $\mathrm{B} := \mathrm{B} - (x \doteq t_1)$
else C := no_match)
else $(\mathrm{B} := \mathrm{B} - (x \doteq t_1)$ & $\mathrm{S} := \mathrm{S} + (x \doteq t_1))$

Liveness

Universal weak fairness

Figure 4-5: Specification with simple guards and a weak liveness property

the existence of equations in the container, but not on properties of those equations. The container object can therefore be represented by a generic type such as a stack, using push and pop for the insert and delete operations, respectively.

Specification 4-3 allowed infinitely repeated stuttering actions, because the same action could be repeatedly "tried" on a single equation. Specification 4-5 records some history information, namely, a recording of actions that have been attempted and failed on certain equations. The single container of equations from the previous specifications is now divided into three, which are best characterized by their invariants:

- E holds equations with the property that the right sides are ground. (This is also true

about E in both of the previous specifications, and relies on the input equation having this form.)

- B holds bindings, which are equations in which the left-hand side is a simple variable, and the right-hand side is ground.
- S is a container of equations in solved form, and is therefore also a substitution.

The *decomposition* and *clash* actions of previous specifications are merged into the *declash* axiom. An equation that appears in B has been placed there by a *declash* action; such an equation can be neither decomposed nor used to generate a clash, because the left-hand side is a variable. Equations in S have been moved there from B by a *consistency* action when they do not create a duplicate assignment. In addition to the three containers, there is a separate state component, C, for recording clashes. In previous specifications, clashes were denoted by setting E to a special (non-container) value; the same technique could be used here, but because E is now represented by three components instead of one, the specification would be messy.

In the initial state, E contains the user's input, the other three containers are empty, and the clash flag is set to *running*. The *declash* axiom defines actions that choose an arbitrary equation in E and, depending on its value, do one of the following: decompose the equation by replacing it by its children, record a head symbol clash by setting C to *no_match*, or move the equation to the container of bindings, B. The *consistency* axiom defines actions that choose a binding from B, and either add the binding to the substitution, S, or if the addition would cause a duplicate variable in S, check the right-hand sides for equality. If the right-hand sides are equal, nothing else is done, but if the right-hand sides are not equal, an inconsistency has been detected so E is set to *no_match*. The explicit test of C in each guard replaces the implicit condition in the previous specifications that the guards were false if E had been set to *no_match*. All legal actions result in a state change, and as with specification 4-4, each action decreases either the number of function symbols or the number of variables in the state.

Adjusting Minimum Granularity

So far, we have demonstrated two ways in which specifications are refined to make them easier to implement: weakening the liveness property and simplifying the transition axioms. Another

factor that should be considered during this phase of program development is the granularity of parallelism. The transition procedures, which implement the transition axioms, are sequential procedures that are scheduled in parallel with one another. Thus, they determine the granularity of parallelism.

If axioms specify actions that are inherently large and expensive, then parallelism will be limited. Alternatively, if actions are small, then a massive amount of parallelism may be possible, but unless there are processors to handle that parallelism, performance will be lost to the overhead of scheduling the transition procedures. Furthermore, if axioms require vastly different amounts of computation, it will be difficult to produce executions that make effective use of processor resources during the entire computation; a long-running transition will be a sequential thread that may limit overall program performance.

To a limited extent, it is better to have axioms that define many small actions than a few large ones, because the transitions axioms define the *minimum* granularity of parallelism. In Section 4.3.4 we show that late in the program development process actions can sometimes be combined to increase the parallelism grain size, but the reverse transformation, splitting actions, is more difficult. The qualification on favoring small actions over large is three-fold:

- Combining actions eliminates some, but not all, of the run-time overhead.
- Overly fine-grained programs are overly complicated, since parallelism is being used in places where sequential code could have been used.
- Programmer time spent implementing fine-grained transition procedures is wasted if the procedures will later be combined.

The matching example does not illustrate the use of restraint on the smallness of grains; the total computation in many actions is a single operator comparison, which could not be any smaller. In the completion program (Chapter 5), however, avoiding overly small grain size is important. For example, matching is a subproblem of completion, but is not parallelized.

In the context of the matching problem, we demonstrate the process of making the minimum grain size smaller. Consider specification 4-5, and recall that *consistency* actions must check whether two ground terms being assigned to the same variable are equal. If the two terms are

large, then it may be desirable to parallelize the *ground_equal* operation within matching, since if the operation is performed sequentially, it takes time proportional to the size of the terms.

There are a number of way in which parallelism could be used in the *ground_equal* operation. It could be treated as a separate problem to be parallelized, but in our approach we require that implementations of transition procedures are sequential, so a parallel implementation of *ground_equal* is not allowed. (We will justify this constraint and discuss generalizations to allow parallelism at multiple levels in Section 4.4.) In this example, exposing the parallelism within *ground_equal* involves adding axioms to the matching specification to describe the actions in testing two terms for equality. Conveniently, term equality is a special case of matching, so the same steps used for parallel matching can be used for parallel equality. One way to change the specification would be to modify the *consistency* axiom so that the two terms being tested for equality are added as an equation to E, where *declash* actions will perform the equality test. But equations added by *consistency* actions have ground terms on both sides, so there would be an unnecessary test in *declash* actions to test whether the left-hand side is a variable. To avoid this test, we add a fourth container I of identities, which are ground equations that need to be checked for equality. We then rename the *declash* axiom to *declash_E*, and add a new axiom *declash_I* that decomposes and checks for clashes in I. The resulting specification is shown in Figure 4-6.

Our goal in developing specification 4-6, was to have each axiom describe actions of the roughly the same complexity. If it takes unit time to check for operator equality and variable equality, then all actions require computation that is independent of the size of terms (although dependent on the arity of operators). Thus, measured on the scale of input size, we have met the design goal. The *consistency* action, however, requires a test to determine whether a variable has already been bound in S, and depending on the choice of representation for S, this cost of this test may not be constant as the size of S grows. In general, performance concerns at this stage of design involve worst case performance for the actions described by transition axioms; the actions should be small enough to provide sufficient parallelism, large enough to avoid excessive overhead, and relatively uniform in cost to simplify scheduling.

State Components

E : container[equation]	% equations
B : container[equation]	% bindings
S : container[equation]	% substitution
I : container[equation]	% identities
C : {running, no_match}	% clash flag

Initially

E = (pattern $\doteq$ target)
B = S = I = $\emptyset$
C = running

Transition Axioms

declash_E
$((p \doteq t) \in \mathrm{E})$ & $(\mathrm{C} \neq \text{no\_match}) \Rightarrow$
if (is_var(p)) **then** $(\mathrm{E} := \mathrm{E} - (p \doteq t)$ & $\mathrm{B} := \mathrm{B} + (p \doteq t))$
else if (p.head = t.head)
then $\mathrm{E} := \mathrm{E} + (p[1] \doteq t[1]) + \ldots + (p[n] \doteq t[n]) - (p \doteq t)$
else C := no_match

consistency
$((x \doteq t_1) \in \mathrm{B})$ & $(\mathrm{C} \neq \text{no\_match}) \Rightarrow$
if $(\exists\ t_2 \mid (x \doteq t_2) \in \mathrm{S})$ **then**
$(\mathrm{B} := \mathrm{B} - (x \doteq t_1)$ & $\mathrm{I} := \mathrm{I} + (t_1 \doteq t_2))$
else $(\mathrm{B} := \mathrm{B} - (x \doteq t_1)$ & $\mathrm{S} := \mathrm{S} + (x \doteq t_1))$

declash_I
$((p \doteq t) \in \mathrm{I})$ & $(\mathrm{C} \neq \text{no\_match}) \Rightarrow$
if (p.head = t.head)
then $\mathrm{I} := \mathrm{I} + (p[1] \doteq t[1]) + \ldots + (p[n] \doteq t[n]) - (p \doteq t)$
else C := no_match

Liveness

Universal weak fairness

Figure 4-6: A fine-grained specification

4.3.3 Transition Procedures for Matching

In this section we describe the third step of the transition-based approach, which involves writing transition procedures. The final specification produced during refinement in step 2 meets the guidelines for directly implementable specifications. In this section we refer to that specification (in Figure 4-6) as *the* directly implementable specification for matching.

Requirements on Transition Procedures

Each transition procedure executes as a single sequential thread, but the procedures must behave correctly when run concurrently with each other. Each procedure has a sequential specification based on a transition axiom, and the set of procedures is required to behave as if they were atomic. We give the precise correctness requirements below.

The first step in implementing the transition procedures is to choose representations for the state components in the specification. The objects need not be the exact type named in the specification, but they must have operations to create initial states from the input, and retrieve answers from the final state. In our examples, most of the state components are implemented by linearizable objects. This is not a requirement of the approach, since in some cases high-level synchronization within the transition procedures will obviate the need for low-level synchronization within the objects. In our experience, linearizability usually, but not always, lead to more highly concurrent transition procedures and more readable code than other strategies.

The set of transition procedures consists of a procedure for each transition axiom in the directly implementable specification, and another procedure used to detect termination. For each guarded command $G \Rightarrow S$ called *name* in the directly implementable specification, a procedure is implemented according to the following interface:

name = **procedure** $(x_1, ..., x_n)$ **signals** (stutter)
ensures: if G is true, either perform S or stutter
if G is false, stutter

The arguments, $x_1, ..., x_n$, are state components (typically those named in either G or S), along with optional *auxiliary* state components used by the implementation. An example of an

implementation that uses such auxiliary objects is given below in the transition procedures for matching. When a procedure invocation stutters, it means that the state is unchanged, and the *stutter* exception is raised. The *stutter* exception will be used when implementing the scheduler to avoid repeatedly calling procedures that do nothing.

The last procedure, called *termination*, is a predicate that returns true exactly when all the guards are false. The *termination* procedure will be used in the scheduler to detect that all computation has been done, so that the scheduler may itself be halted.

termination = **procedure** $(x_1, ..., x_n)$ **returns** (bool)
ensures: if all guards are false, return true
if any guard is true, return false

Although the transition procedures name multiple objects (that can be mutated by the procedures), this is merely a convenience to document which state components are used by which transition procedures; the transition procedures can be viewed as operations on the single data type that constitutes the program's state. Using this view, the correctness notions of Chapter 2 are immediately applicable.

The set of transition procedures must be linearizable, non-stopping, and *productive*; the last of these is defined below. The procedures have no interference specifications, and the specifications are non-waiting, so the scheduler can safely invoke any procedure at any time. The non-waiting property was realized by extending the guarded commands to allow stuttering, since a guarded command denotes a waiting operation with the *when* clause given by the guard. The specifications admits stuttering in any state, even when the procedure is *enabled*, i.e., the guard of the corresponding transition axiom is true. The reason for allowing stuttering of enabled procedures is so that procedure can abort (by stuttering) if it determines that completing the operation normally would be too expensive, e.g., because needed locks are being held by some other operation.

Because the transition procedure specifications allow stuttering in any state, an additional liveness property is placed on the transition procedures to preclude implementations that always stutter. The *stutter-free form* a sequential history H (as defined in Chapter 2), is the subhistory with all maximal finite sequences of stuttering operations removed. A linearizable history H is *productive* if there is a linearized history H' of H such that its stutter-free form is correct.

Thus, productivity allows for finite sequences of superfluous stuttering, but if there is an infinite sequence of stuttering operations, the stuttering procedures must not be enabled.

As noted in Chapter 2, a consequence of using these correctness conditions is that certain producer/consumer-style relationships cannot exist between procedures. For example, if a consuming procedure were designed to block and wait for data from the producer, then executions in which no producer is running would be blocked. Instead, the consuming procedure is implemented to check for data to be consumed, and return if none is available; it is then up to the invoking process (in our case the scheduler) to determine whether the consuming procedure should be retried or not. This program structure will occur frequently in programs developed with our method.

There is also an informal performance requirement on the transition procedures: they must exhibit real concurrency. Ideally, a procedure invocation would not be slowed by other concurrently executing procedures, but, because there may be synchronization for access to common data, this goal cannot always be met. The level of concurrency allowed by these procedures will limit the amount of parallelism in the overall program, so at a minimum, it should be faster to run two procedures concurrently than to run them serially, one after the other. This disallows the obviously correct implementation in which each procedure executes entirely within a single critical region, since it will result in a program with almost no real parallelism. To obtain a high degree of concurrency, we use the heuristic that critical regions should be short, infrequent, and shared with few other procedures.

Transition Procedure Implementations

The state components in the directly implementable specification consist of four containers of equations, and a flag to denote a detected clash. As a first-cut design, we make equations immutable objects, and the containers that hold them mutable, linearizable queues. (One of the containers will be changed below, when we are better able to motivate the choice.) The clash flag, C, is simply a shared memory location.

Writing transition procedures for the directly implementable specification would be straightforward if they were only to be invoked serially. Consider, for example, the implementation of *declash_I* and *termination* in Figure 4-7. In any serial invocations, these procedures behave

```
declash_I = procedure (I: container[equation], C: location) signals (stutter)
        if (C = no_match) then signal stutter
        e: eqn := dequeue(I)
                except when empty: signal stutter
        if (head(e.left) = head(e.right))
            then
                for (i := 1) to arity(e.left) enqueue(I, (e.left[i] ≐ e.right[i]))
            else C := no_match
    end declash_I

termination = procedure (E, I, B: container[equation], C: location) returns (bool)
        if (C = no_match) then return(true)
        if ((size(E) = 0) & (size(B) = 0) & (size(I) = 0))
            then return(true)
            else return(false)
    end termination
```

Figure 4-7: Matching procedures that work serially

according to their specifications, but when invoked concurrently they do not. We demonstrate the problem by looking at an execution that starts from the (reachable) program state in which E and B are empty, and I has the single equation $(f(a,b) \doteq f(a,b))$. The following interleaving of operations is possible: *declash_I* removes the last equation, $(f(a,b) \doteq f(a,b))$, from I, so $I = B = E = \emptyset$; *termination* tests the three containers I, B, and E, finds them empty, and returns *true*; *declash_I* inserts the two child equations, $(a \doteq a)$ and $(b \doteq b)$ into I and returns. This execution is not linearizable, because in a linearized execution, *termination* would have to be executed either before or after *declash_I*. I is non-empty in both the state before and the state after *declash_I* executes, so in either state the guard for *declash_I* guard is true. Thus, *termination* should have returned *false*.

One way to repair the implementation would be to add a critical region to *declash_I* starting before the *dequeue* and ending after the *enqueue*, and add the same critical region around the size test in *termination*. The effect of this would be to serialize two concurrent instances of *declash_I*, as well as concurrent instances of *declash_I* and *termination*. Implementations of the other two transition axioms, *declash_E* and *consistency*, would have similar problems, so those procedures would also have similar critical regions. The three procedures implementing

transition axioms could protect their critical regions with distinct locks, all acquired by the *termination* procedure. These procedures could safely be invoked with any degree of concurrency, but at most one instance each of *declash_I*, *declash_E*, and *consistency* would actually execute concurrently. Because the critical regions within the transition procedures prevent concurrent access to the containers, this implementation would have the advantage that none of the containers require internal synchronization—any serial queue implementation would do. Expecting only a small constant speedup in matching, we implemented these procedures as described. The performance was even worse than expected: the synchronization overhead resulted in a matching program that was significantly slower than a sequential implementation on all inputs. This led us to the development of a set of transition procedures with higher throughput.

Instead of adding long critical regions to the code, a better implementation uses linearizable container types and incorporates auxiliary variables to achieve linearizability at the level of the transition procedures. These procedures, shown in Figure 4-8, resulted in the best overall performance of any we considered.

The transition procedures in Figure 4-8 are linearizable, non-stopping, and productive. In Figure 4-7, the problem is that the termination condition (I, E, and B being empty) could temporarily be true while equations were being moved between containers or replaced by child equations. Because the transition procedure specifications admit stuttering even when the guard is enabled, it is acceptable for actions to see empty containers and therefore stutter; the only problem is that *termination* must not return true when a container is about to be refilled. In Figure 4-8, three auxiliary objects, *E_size*, *B_size*, and *I_size*, are added to the state to represent the size of the three containers being tested. The objects hold the exact size of the corresponding container when no transition procedure is actively using that container, but when such a procedure is in progress, the objects may hold an out-of-date value. The auxiliary objects have the flavor of history variables that are useful in correctness proofs of parallel programs [OG76], but the auxiliary objects in our implementation actually exist.

The state in Figure 4-8 uses four different linearizable data types: queues of equations (*eq_queue*), *accumulators*, memory *locations*, and *mappings*. The state components are given here with their corresponding types–the type of C is a *location* that may hold one of the named values:

```
declash_E = procedure (E, B: eq_queue, C: ptr[int],
                       E_size, B_size: accum) signals (stutter)
    if (C = no_match) then signal stutter
    e: eqn := dequeue(E)
              except when empty: signal stutter
    if is_var(e.left) then
        accum$add(B_size, 1)
        accum$add(E_size, −1)
        enqueue(B, e)}
    else if (head(e.left) = head(e.right)) then
        {accum$add(E_size, arity(e.left) − 1)
        for (i := 1) to arity(e.left) enqueue(E, (e.left[i] ≐ e.right[i]))}
    else C := no_match
  end declash_E

consistency = procedure (B, I: eq_queue, S: substitution, C: ptr[int],
                         B_size, I_size: accum) signals (stutter)
    if (C = no_match) then signal stutter
    e: eqn := dequeue(B)
              except when empty: signal stutter
    assign(S, e.left, e.right)
        except when already_bound(t):
                    {accum$add(I_size, 1)
                    accum$add(B_size, −1)
                    enqueue(I, (e.right ≐ t))
                    return}
    accum$add(B_size, −1)
  end consistency

declash_I = procedure (I: eq_queue, C: ptr[int], I_size: accum) signals (stutter)
    if (C = no_match) then signal stutter
    e: eqn := dequeue(I)
              except when empty: signal stutter
    if (head(e.left) = head(e.right)) then
        {accum$add(I_size, arity(e.left) − 1)
        for (i := 1) to arity(e.left) enqueue(I, (e.left[i] ≐ e.right[i]))}
    else C := no_match
  end declash_I

termination = procedure (C: ptr[int], E_size, B_size, I_size: accum) returns (bool)
    if (C = no_match) then return(true)
    if ((accum$read(E_size) = 0) & (accum$read(B_size) = 0) & (accum$read(I_size) = 0))
        then return(true)
        else return(false)
  end termination
```

Figure 4-8: Transition procedures for matching

E, B, I: eq_queue

E_size, B_size, I_size: accum

S: mapping

C: location[{running, no_match}]

In writing the transition procedures, lower level data types are assumed to be linearizable. Invocations of *enqueue* and *dequeue*, for example, can be viewed as indivisible operations, although they can, and in our implementation do, execute concurrently with one another. The *mapping* abstraction was specified and implemented in Chapter 3. Recall that *Mappings* are multi-ported objects, and the *insert* operation interferes with other instances of *insert* on the same port. This interference requirement is met by making the state multi-ported as well, and giving each thread in the scheduler a different port. This level of detail is not evident in the code given below.

The *termination* procedure checks the values of *E_size*, *B_size*, *I_size*, rather than the size of each container. In the initial state, *E_size* is one, and *B_size* and *I_size* are zero. Each of these *size* objects is updated in a delayed fashion by the procedures accessing the corresponding container. For example, when the *declash_E* procedure removes an element of E that needs to be decomposed, the *E_size* value is not updated until after the need to decompose has been determined, and after the number of child equations is known. At that time, one less than the number of children is added to *E_size*; this single atomic update shadows the actual size of E, which is first decremented by one and then incremented by the number of child equations. Other *declash_E* actions may observe an empty E, and therefore stutter, but concurrent *termination* actions will not observe a zero value for *E_size*. Similarly, when an equation is moved from one container to the next, for example, from E to B by a *declash_E* action, the *B_size* accumulator is incremented before the *E_size* accumulator is decremented. Because the three sizes are tested in the order *E_size*, followed by *B_size*, followed by *I_size*, the entire expression will only be true if all of the sizes are zero, and will remain zero. We discuss this point in more detail in Section 4.3.5.

Testing and Debugging Transition Procedures

Given the complicated nature of the transition procedures, correctness arguments of the kind given in Section 4.3.5 are important. However, there are generally easier methods for finding programming bugs than development of a correctness proof. The transition-based method supports modular debugging and testing, in addition to correctness arguments. To debug a set of transition procedures, the easiest first step is to run them sequentially, ridding them of bugs that have nothing to do with parallelism. This technique is not usable for arbitrary parallel program modules, because, for example, one procedure might depend on the concurrent execution of another procedure to make progress. The technique can be used with transition procedures because the non-stopping requirement rules out such dependencies.

A second debugging technique is to test subsets of the transition procedures. For example, multiple instances of the same procedure typically share data in a nontrivial way, so running some number of instances of the same procedure concurrently can locate bugs within that procedure. In the matching program, for example, the state can be initialized with ground equations in I, and every other container empty, and the *declash_I* procedure can be tested alone or with the *termination* procedure. Ideally, one would examine all combinations of procedures, on all input values, for every interleaving of the lower level operations; this would literally test the condition, "every concurrent execution must be equivalent to some sequential one." Given the impossibility of this task, we settle for a few interesting executions; combinations are limited to two or three concurrent invocations, input values are chosen as they would be for testing sequential programs, and various interleavings are forced by controlled simulation or, haphazardly, by inserting delays into the code.

Testing and debugging parallel procedures is still a difficult process, but in this case is greatly aided by the modularity of the design. In testing the transition procedures for matching, it is sufficient to consider interleavings (rather than concurrent executions) of the lower level operations, since the operations can be tested separately during development of the data abstractions.

From linearizability it follows that any concurrent execution is equivalent to some sequential one, and combined with the partial correctness of transition axioms, a powerful property of the transition procedures follows: any (possibly concurrent) execution of the transition procedures

is safe, i.e., does not compute an incorrect answer, and if finite, can be extended to one that computes the correct answer. This property extends the partial correctness of the transition axioms to concurrent executions, and has practical ramifications as well; while the axioms only describe transitions abstractly, the procedures implement them. These conditions gives us a great deal of flexibility when designing a scheduler, since if the transition procedures have been implemented correctly, the scheduler cannot create race conditions or deadlock between them.

4.3.4 A Scheduler for Matching

The final step of the transition-based approach is an implementation of a scheduler for the transition procedures. This section describes our strategy for implementing schedulers in general, and some schedulers for the matching problem in particular. Since performance tuning is a significant part of scheduler development, performance numbers for matching are included here.

Scheduler Requirements

Before presenting the scheduler step of the approach, we define some terminology and discuss the general problem of scheduling multiprocessor programs.

A *thread* is a lightweight process. Each thread has its own stack of procedure invocations, but there is a single object heap common to all threads in a program. We assume our programming language has a construct such as *fork*, for starting a new thread. The *fork* primitive takes a procedure name and a list of arguments, and it has the effect of starting a new thread to execute the procedure. Because the unit of work associated with a thread is a procedure, one might be misled into thinking that any procedure that can be evaluated in parallel should be "forked off." However, when the number of threads is much greater than the number of processors, or the threads are short-lived, the overhead of creating separate stacks and switching between thread contexts is prohibitive. A second problem with having more threads than processors is that scheduling of threads is done by the system scheduler; application-specific scheduling usually yields much better performance [RV89, JW90].

The solution to both problems is to start a fixed number of threads at the beginning of a program's execution, and to schedule tasks onto these threads. Examples of this approach

include tuple spaces [CG89], work crews [RV89], and supervisors [JW90]; the approaches differ in the kinds of synchronization allowed between tasks, and the underlying scheduling algorithm. A *task* is a procedure invocation (a procedure plus input arguments) that is guaranteed to terminate.

A scheduler for a set of transition procedures is a multi-threaded program that concurrently invokes the procedures. The correctness requirement on a scheduler is that it must guarantee the liveness property of the directly implementable specification from step 2. Thus, the distinction between safety and liveness in the specification translates to a separation between requirements on the transition procedures and requirements on the scheduler in the implementation.

Because the directly implementable specification has a weak liveness property, it is not difficult to meet the correctness requirement of the scheduler. Most of the work of producing a scheduler is performance tuning, since the effect of scheduling decisions on performance is often difficult to predict. The design process can be seen as largely trial and error, wherein a scheduler is designed, implemented, and performance tested, and if the result is unsatisfactory, the entire process is repeated. Without the weak correctness requirement on the scheduler, this development cycle would be prohibitive.

For simplicity, we assume that the number of available processors is a known quantity, i.e., it is either known at compile time, or can be quickly determined at run-time before the application begins the main computation phase. Once determined, this quantity remains constant. A scheduler spawns a fixed number of threads equal to the number of available processors, and each thread runs a copy of scheduling code; it is typical, although not necessary, that the code run by each thread is identical.

Our assumption that there are at least as many processors as threads is an over simplification, since the operating system may, at any time, usurp a processor for its own use. If a thread is de-scheduled while holding a lock, the performance impact can be significant. A different operating system interface could prevent this problem by allowing the operating system scheduler to communicate with the application scheduler; if communication is sufficiently limited, this interface need not incur the full cost of operating system scheduling [ABLL90]. Using our method, program performance depends on the number of threads being no greater than the number of processors, but program correctness does not depend on this assumption. Correct-

ness of the scheduler, and therefore the program, does depend on fairness of the underlying system scheduler in scheduling threads.

Assumptions and Implications

Application-specific scheduling is more efficient than relying on a system-defined scheduler for a number of reasons. At the system level there is no information about the kinds of tasks being executed in parallel, so worst-case assumptions are often made. For example, when one task can depend on the execution of another, the scheduler preempts tasks to guarantee that each gets an opportunity to execute. The direct cost of switching contexts, along with the indirect costs due to lost cache context, can make such fair scheduling strategies prohibitive. Contexts can also be quite large on some machines, stressing system resources beyond their capacities. In general, application-specific scheduling can use information about the program to determine points at which contexts are small, and therefore context switches are cheap. An additional benefit follows from our approach to building parallel programs. Because of the correctness condition on transition procedures, there can be no dependencies between procedures, so each is run to completion once it is scheduled.

A second advantage of application-specific scheduling is the ability to use knowledge about which tasks should be favored to get an efficient schedule. For example, some tasks may generate more work to do in parallel, thereby making it possible to keep more processors busy finding a solution. Alternatively, some tasks may lead to useless work; for example, speculative parallelism can lead to redundant computation. In our programming model, in which each transition procedure is the implementation of a guarded command, there is an advantage to executing a procedure for which the guard is true.

Because all of the transition procedures have non-waiting specifications and non-stopping implementations, in any execution there is some procedure that will eventually return. By induction, we can prove that if the scheduler stops invoking procedures, all invoked procedures will eventually terminate. However, the scheduler may invoke transition procedures that will simply stutter, and there is no run-time mechanism that the scheduler can use to determine whether or not a transition procedure is going to stutter. In defining this interface between transition procedures and the scheduler, we are trading context switching overhead (of a sched-

```
match_state = record[E, B, I: eq_queue, S: substitution, C: ptr[int],
                     E_size, B_size, I_size: accum]
sched_loop = procedure (m: match_state)
        while¬termination(m.C, m.E_size, m.B_size, m.I_size)
            {declash_E(m.E, m.B, m.C, m.E_size, m.B_size)
            consistency(m.B, m.I, m.S, m.C, m.B_size, m.I_size)
            declash_I(m.I, m.C, m.I_size)}
  end sched_loop
scheduler = procedure (m: match_state)
                ⋮
        for (i := 1) to process_limit do
            fork(sched_loop, m)
                ⋮
  end scheduler
```

Figure 4-9: Distributed round-robin scheduler

uler that allows procedures to block) with the cost of stuttering procedure invocations. The judiciousness of this choice depends on the possibility of defining schedulers that can predict efficient executions based on static information about the application and dynamic information about prior stuttering invocations.

Scheduler Implementation

In the directly implementable matching specification (Figure 4-6), there is a single liveness conditions, universal weak fairness. We begin by presenting an obviously correct scheduler for matching, one in which each thread alternates among the transition procedures in a round-robin style, and halts whenever the *termination* procedure returns true. The code for all of the scheduling threads is given by the *sched_loop* procedure in Figure 4-9. The scheduler is distributed, as demonstrated by the fragment of the *scheduler* procedure.

In general, it is difficult to predict *a priori* what scheduler will yield the best performance. Performance models of parallel machines are not detailed enough to allow accurate predictions about task handling overhead, contention for application-level locks, and contention for system resources such as the memory bus. While more precise performance models would be helpful,

it is not clear that a model containing all of these details would be usable. Therefore, the scheduling step in the transition-based approach allows for a significant amount of performance tuning. Not all performance problems can be addressed by changing the scheduler; for example, reducing contention within a data structure is often done by re-designing the data structure. However, we describe some transformations to the scheduling algorithm that improve the performance. The transformations themselves are useful in other applications, and the effect they have on performance is significant.

Performance Tuning

Before describing the scheduler transformations, some comments about the performance numbers are timely. All performance numbers are taken on a Firefly multi-processor [TSJ87], using up to five CVAX processors. The input to the matching program is a pair of terms, and in discussing performance, we work with a set of inputs that characterizes various classes. The inputs are not typical of real applications of matching, because they are too large (tens of thousands of function symbols in each term). For small input terms, the cost of starting threads exceeds the cost of performing the matching operation.

Each of the inputs in the test suite is characteristic of a class of inputs. See Figure 4-10 for an example of performance statistics; the inputs are named in the left-most column.

- *BalGround* contains two identical ground terms. Since there are no variables in the input, this example will not use any *consistency* or *equality* steps. The terms are perfectly balanced trees. This should be the easiest input to parallelize.

- *BalRepeat1*—*BalRepeat3* are also balanced, but each contains variables in the pattern. *BalRepeat1* has the fewest variables, while *BalRepeat3* has the most (a variable at every leaf). Each variable in the pattern is repeated, so these inputs test the effect of collisions in the mapping abstraction.

- *BalDiffer* is balanced with variables, but the variables are not frequently repeated, so collisions are unlikely, but the answer mapping grows large.

- *UnbalDiffer* and *UnbalRepeat* are unbalanced trees. *UnbalRepeat* has frequently repeated variables, while *UnbalDiffer* does not. The unbalanced terms test processor utilization

	absolute 1 thread	speedup 1 : 2	speedup 1 : 3	speedup 1 : 4	speedup 1 : 5	speedup seq : 5
BalGround	113 ms	1.56	1.91	2.12	2.20	0.55
BalRepeat1	803 ms	1.81	2.30	2.62	2.78	0.62
BalRepeat2	833 ms	1.82	2.26	2.59	2.80	0.66
BalRepeat3	908 ms	1.76	2.30	2.70	2.83	0.50
BalDiffer	929 ms	1.76	2.36	2.70	2.85	0.59
UnbalRepeat	301 ms	1.74	2.22	2.49	2.68	0.56
UnbalDiffer	890 ms	1.68	2.24	2.66	2.73	0.52

Figure 4-10: Performance of the round robin scheduler on stable inputs

with heterogeneous input data.

- *ClashLT*, *ClashRT*, *ClashLB*, and *ClashRB* all contain function symbols in the pattern that clash with the corresponding symbol in the target. The clash symbols occurs in a different positions in each example. The positions are, left top, right top, left bottom, right bottom, respectively. These inputs test processor utilization for data that results in performance instability.

Figure 4-10 contains data for the matching program using the scheduling algorithm in Figure 4-9, which was one of the first schedulers we implemented. The first column of numbers is the absolute performance of the program using only one thread. The middle four columns are the speedup of the two, three, four, and five threads, relative to the one thread version (i.e., the parallel program using only one scheduler thread). The last column is the speedup of the five thread parallel program, relative to a sequential implementation. The sequential program was implemented before the parallel one, and and is optimized for sequential execution. It serves as a reality check to ensure we are not comparing speedups of a slow parallel program against itself.

This set of numbers includes only the stable inputs, i.e., those without clashes. Even for these well-behaved inputs, the performance of this version of matching is discouraging. Not only does the sequential program consistently out-perform the parallel one, but when comparing the parallel program against on one thread, the speedups never exceed a factor of 3.00. These results led us to consider variations on the scheduler.

	absolute 1 thread	speedup 1 : 2	speedup 1 : 3	speedup 1 : 4	speedup 1 : 5	speedup seq : 5
BalGround	1635 ms	1.97	2.04	3.71	3.92	3.62
BalRepeat1	12213 ms	2.02	2.23	3.91	4.04	3.68
BalRepeat2	14028 ms	2.02	2.32	3.86	4.38	3.65
BalRepeat3	16888 ms	1.92	2.74	3.37	3.92	2.46
BalDiffer	20235 ms	1.90	2.69	3.47	4.06	2.33
UnbalRepeat	17490 ms	1.90	2.75	3.40	3.93	2.38
UnbalDiffer	4357 ms	1.96	2.90	3.88	4.79	4.77

Figure 4-11: Performance of a coarse grained scheduler on stable input

The easiest change that can be made to the scheduler is to use a different execution order for the transition procedures. For example, each transition procedure can be embedded in a *while* loop that repeatedly calls the transition procedure until it stutters. One transition can also be prioritized over another by invoking it with greater frequency. We tried a number of these reordering schemes, but none had a significant impact on performance. The round robin scheduling order is not the best one for all applications. For example, the completion program in Chapter 5 prioritizes some transitions over others, and changing the order has a significant effect on performance.

The control structure in Figure 4-9 is not the only thing that determines the schedule. The choice of data structure for representing containers also has an impact. The transition axioms do not specify any order on the elements in containers, so stacks or priority queues or other container-like structures can be used in place of queues. The order in which elements are chosen from the containers does not change the order in which transition procedures are invoked, but does change the order in which data is used. We experimented with stacks as well as queues, but again there was no positive change in performance. (When stacks are used, the parallel program is much closer to the depth-first sequential one, so the inputs with clashes are no longer wildly different.) Again, the completion program will make use of a different data structure, a priority queue, which is essential to that program's performance.

Scheduler transformations involve changes to the code in Figure 4-9, and sometimes to the transition procedures. A scheduling change that significantly improves performance is to increase the grain size of the tasks being scheduled, and this involves changing the transition

	absolute 1 thread	speedup 1 : 2	speedup 1 : 3	speedup 1 : 4	speedup 1 : 5	speedup seq : 5
ClashLT	4 ms	0.61	0.39	0.33	0.28	0.10
ClashRT	236 ms	2.03	20.74	16.65	14.96	13.43
ClashLB	234 ms	2.04	1.86	1.91	1.91	1.75
ClashRB	457 ms	2.03	2.04	3.66	3.57	3.42

Figure 4-12: Performance of a coarse grained scheduler on unstable input

procedures. To change the grain size, the transition procedures must, in some cases, invoke other transition procedures directly. For example, the *declash_E* procedure calls itself recursively some number of times, before new equations are added to the container. Figure 4-11 gives the performance results for clash-free inputs using the algorithm with larger grain size.[2]

If we consider the unstable inputs, instead, the speedups become very erratic. Results for the inputs with clashes are shown in Figure 4-12. Because we want an efficient sequential program as the baseline, a depth-first evaluation order is used in that program. The parallel program, on the other hand, has a natural breadth-first order that comes from the parallel evaluation of multiple children of a single node. The parallel version is not entirely breadth-first, however, because when the grain size is increased, large sub-problems within matching are done sequentially, and therefore depth-first. In matching it is easy to see why placement of clashes can result in super-linear speedups, or case in which the parallel program is slower than the sequential one. Such anomalies should be expected in problems with performance instability, but unfortunately, they are not always easy to explain. With completion, for example, we observed similar, although less extreme, performance anomalies; the completion process is complex enough that the anomalies are harder to explain.

4.3.5 Correctness Arguments

This section presents the correctness arguments for two steps in the matching example. First, we show that each of the refined specifications in Section 4.3.2 satisfies the original one. We then give some of the key arguments for the correctness of the transition procedure implementations.

[2]The coarse-grained scheduler was run on much larger examples than were feasible with the fine-grained scheduler. Thus, the absolute performance should not be compared between the two sets of numbers.

The exercise of presenting these proofs is instructive: it demonstrates what should be proved about programs developed using the transition-based approach, and some of the techniques that are useful in finding the proofs. The correctness arguments for matching use standard *refinement mapping* proof techniques [AL88].

Correctness of the Transition-Axiom Specifications

In Section 4.3.1 we argued that specification 4-3 solves the matching problem. To show that the three specifications 4-4 through 4-6 also solve the matching problem, we can either make a direct argument showing that each specification solves matching, or an indirect argument that each specification satisfies specification 4-3. The latter approach is advantageous for proving the correctness of large complex algorithms, because the proofs can be done in stages.

An interesting aspect of the correctness arguments is the existance of parallels between the proof effort and the specification effort. Each of the transition-axiom specifications was developed with a particular goal in mind, and the effect of achieving that goal was to make a particular piece of the satisfaction proof non-trivial. Specification 4-4 replaced the strong liveness property of specification 4-3 with something weaker, so proof of the liveness property constitutes the bulk of the satisfaction argument between them. Specification 4-5 replaced the transition axioms of specification 4-4 with axioms having simpler guards; the proof of the safety property is the more difficult part, and it relies on proving an invariant that shows the weaker guards to be sufficient. Specification 4-6 is a classic *refinement* of specification 4-5, because a *consistency* action in the first is replaced by a sequence of smaller actions in the second; the proof makes use of *history variables* [OG76], a proof technology known to be useful for such refinments.

The usual notion of satisfaction between transition axiom specifications requires preservation of whatever states (or actions) are externally visible. For example, in the state-based approach of [AL88], certain components of the state are designated as *external*; one specification is said to *satisfy* another if for any live execution of the first, there is a live execution of the second such that the histories restricted to external states are identical (modulo finite stuttering actions). The reader is referred to [AL88] for a precise definition of satisfaction; similar notions of satisfaction are defined for the I/O Automata model [LT87] and for Unity programs [CM88].

Technically, our series of specifications, 4-4 through 4-6 do not satisfy specificaton 4-3 because the initial and final states differ in minor ways. For example, the container E in specification 4-4 holds the user's input in the initial state and the answer in the final state. If E is designated as the external state of the specification, then the intermediate values of that E takes on mid-execution, which are neither the input nor the answer, will be observable to the user. Furthermore, in specification 4-5, the input starts in container E, but the answer ends up in container S, so neither of these will have the same sequence of values as the container E in specification 4-4.

The problem can be solved by adding an additional state component, for example, a container called *input-output*, in which the input and output of matching are stored. Each of the specifications could be augmented to admit an action that copies the input into the expected component, and as the last action, replaces the value of *input-output* with the answer. Thus, *input-output* is the external state. The user only sees two values in *input-output*, the input and the output, and (assuming our specifications compute the right answer) the output will always be the matching substitution for the input.

Because this technical solution has little value in understanding the matching specifications, or how they relate to one another, these augmented specifications and their correctness arguments are not given. Instead, we show the important pieces of the correctness arguments, i.e., the invariants within and abstraction functions between specifications.

An *abstraction function* maps states of a refined specification to states of the original (abstract) specification. Abstraction functions have been used extensively in the context of abstract data types for sequential programs [LG86], and by others in reasoning about parallel and distributed programs [AL88, LT87, HW90].[3] In our use, the state machine part of a specification is viewed as an abstract data type, where the state components contain the type's values, and the transition axioms specify the type's operations. The principle difference between reasoning about most sequential data type implementations and reasoning about transition-axiom specifications is that the latter has a liveness property as part of the proof obligation.

Let $\mathcal{A}$ be an abstraction function on some set of states S. If $\langle s_1, s_2, ... \rangle$ is an execution of

[3] [LT87] and [HW90] use the related notion of a possibilities mapping, which maps a single to state to a set of states.

states in S, then its *abstracted execution* is the sequence $\langle \mathcal{A}(s_1), \mathcal{A}(s_2), ... \rangle$. The *stutter-free form* of an execution is another execution constructed by removing all maximal finite sequences of stuttering actions. Given two specifications S_1 and S_2, we say S_1 is a *refinement* of S_2 by $\mathcal{A}$, if for every execution allowed by S_1, the stutter-free form of the abstracted execution is allowed by S_2.

To prove that S_1 is a refinement of S_2, there are three steps [AL88]:

1. Define an abstraction function, $\mathcal{A}$, from the states of S_1 to states of S_2.

2. Prove that S_1 ensures the safety property of S_2. First, show that for all initial states s of S_1, the state $\mathcal{A}(s)$ is an initial state of S_2. Second, show that for every pair of states (s_1, s_2) that is a legal action of S_1, the action $\langle \mathcal{A}(s_1), \mathcal{A}(s_2) \rangle$ is a legal action of S_2.

3. Prove that for any live execution of S_1, its abstracted execution meets the liveness property of S_2. These proofs are typically done by induction.

We show that each of the specifications in this section are refinements of the original specification. In general, we may need notions other than refinement for showing satisfiability, for example, to show that a coarse-grained specification satisfies a finer-grained one as is the case in Chapter 5. In this chapter, refinement is sufficient, and since it is a transitive notion, we show that each specification is a refinement of the one presented before it.

Lemma 2 *Specification 4-4 is a refinement of specification 4-3.*

For a proof of Lemma 2, let the abstraction function, $\mathcal{A}$, be the identity function. The safety property is obviously ensured, since both specifications have the same initial conditions, and every action allowed in 4-4 is also allowed in 4-3. (The only difference between the sets of actions is that specification 4-3 allows stuttering actions that are not allowed by specification 4-4).

Proving the liveness property is more interesting. First, we argue that every execution terminates by showing that it must reach a state in which no axiom is enabled. Assume that a state is a container of equations or is a special value *no_match*, as specified in 4-4. Given a state s, let $vars(s)$ be the number of variables in s, and $ops(s)$ be the number of function symbols in s; in both cases multiple instances are counted multiple times. For example, if s is the container $\{(f(x) \doteq f(c)), (g(x) \doteq f(c))\}$, where x is a variable and c is a constant, then

$vars(s) = 2$ and $ops(s) = 6$. If s is equal to *no_match*, then $vars(s) = 0$ and $ops(s) = 0$. Let $\prec$ be a partial ordering on states defined by $s_1 \prec s_2$ if and only if one of the following holds:

- $vars(s_1) < vars(s_2)$, or
- $vars(s_1) = vars(s_2)$ and $ops(s_1) < ops(s_2)$.

Any legal action of specification 4-4 results in a state change, and the new state is strictly smaller (by $\prec$) than the old one. Consider a legal action $a = (s_1, s_2)$ and observe that $s_2 \prec s_1$:

1. If a is a *clash* action, then $ops(s_1) \geq 2$, and $vars(s_2) = ops(s_2) = 0$.
2. If a is a *decomposition* action, then $vars(s_2) = vars(s_1)$ and $ops(s_2) = ops(s_1) - 2$.
3. If a is a *consistency* action, then $vars(s_2) = vars(s_1) - 1$.

Therefore, every execution eventually reaches a state in which none of the axioms is enabled. Implicitly, this argument shows why the specification did not require a stronger fairness property, such as fairness between the axioms. Having shown that any execution eventually reaches a state in which all guards are false, we argue that in a final state either E is equal to *no_match* or E is in solved form. Assume E is not equal to *no_match*. When the *decomposition* and *clash* guards are false, E contains only equations with variables on the left-hand side, and when, in addition, the *consistency* guard is false, each variable occurs at most once in E. Therefore, E is in solved form.

Lemma 3 *Specification 4-5 is a refinement of specification 4-4.*

Lemma 3 states a second refinement relation on specifications. In the proof of this lemma, the abstraction function, $\mathcal{A}$, maps a state of specification 4-5 (four equation containers and a clash flag) to a state of specification 4-4 (a single object that is either an equation container or *no_match*).

$$\mathcal{A} : (\text{container[equation]})^3 \times \{\text{running}, \text{no\_match}\} \rightarrow (\text{container[equation]} + \text{no\_match})$$
$$\mathcal{A}(\text{E}, \text{B}, \text{S}, \text{C}) \equiv \textbf{if} \quad (\text{C} = \text{no\_match}) \ \textbf{then} \ \text{no\_match} \ \textbf{else} \ \text{E} \cup \text{B} \cup \text{S}$$

Expanding on the view that a state machine is simply an abstract data type, we note that the invariants given earlier on state components E, B, and S are the analog of representation

invariants in abstract data type implementations. Let $s = (E, B, S, C)$ be a state of specification 4-5, and $\mathcal{I}$ be the conjunction of the state component invariants.

$$\begin{aligned}\mathcal{I}(\langle \mathrm{E}, \mathrm{B}, \mathrm{S} \rangle) \equiv ((e \in \mathrm{E} \Rightarrow \mathrm{is\_ground}(e.\mathrm{right})) \;\&\\ (e \in \mathrm{B} \Rightarrow (\mathrm{is\_var}(e.\mathrm{left}) \;\&\; \mathrm{is\_ground}(e.\mathrm{right}))) \;\&\\ (\mathrm{S\ is\ in\ solved\ form}))\end{aligned}$$

In general, if we add verification of the invariants to the proof obligations, the abstraction function need only be defined on states in which the invariants hold. In this example, the abstraction function, $\mathcal{A}$, happens to be defined on any state in which values are of the appropriate type; nevertheless, the invariants must be shown as part of the safety proof.

In proving the safety property, we start with the argument that initial states map to initial states. In specification 4-5, an initial state, s, has $C = \mathit{running}$, and all containers empty except for E, which contains the user's input. Thus, $\mathcal{A}(s)$ is the container holding the input equation, which is an initial state of specification 4-4. Next, we show that any legal action, $a = (s_1, s_2)$, of specification 4-5 maintains the invariant $\mathcal{I}$ and meets the safety criterion on actions. Assuming $\mathcal{I}(s_1)$, we show $\mathcal{I}(s_2)$ and show that $(\mathcal{A}(s_1), \mathcal{A}(s_2))$ is an action of specification 4-4. We abuse our notation and write $\mathcal{A}(a)$ for $(\mathcal{A}(s_1), \mathcal{A}(s_2))$. Let $s_1 = (E_1, B_1, S_1, C_1)$ and $s_2 = (E_2, B_2, S_2, C_2)$.

- If a is a *declash* action, then there is an equation e in E_1, such that either e is not in E_2, or e has different head symbols on the two sides, and $C_2 = \mathit{no\_match}$. If $e.\mathit{left}$ is a variable, then $\mathcal{A}(a)$ is a stuttering action, and e is moved from E_1 to to B_2. The invariant that B contains bindings is maintained since e is a binding. If $e.\mathit{left}$ is not a variable, the abstract action depends on the head symbols of the right and left-hand sides. If e's head symbols are equal, then $\mathcal{A}(a)$ is a *decomposition* action, while if the symbols are not equal, then $\mathcal{A}(a)$ is a *clash* action; in either case the only affected container is E, so $\mathcal{I}$ is maintained.

- If a is a *consistency* action, then there is an equation, e_1, in B_1 such that either e_1 is not in B_2, or $C_2 = \mathit{no\_match}$. By the invariant on B_1, the left-hand side of e is some variable x so there are two cases depending on whether or not x occurs in S_1. If not, adding e_1 to S_1 does not invalidate $\mathcal{I}$. If so, then there is another equation e_2 in S_1 with x as a

left-hand side. Both e_1 and e_2 have ground terms on the right, so *ground_equal* can be used to test for equality. In either case, $\mathcal{A}(a)$ is a *consistency* action.

The liveness part of the refinement argument is nearly trivial. Both specifications 4-5 and 4-4 have the same stated condition, universal weak fairness. Recall that this condition asserts that executions can only end in states in which all axioms are disabled. Let s be a state of specification 4-5 such that all guards of 4-5 are false. It is sufficient to show that for any such s, all guards of 4-4 are false in $\mathcal{A}(s)$. There are two cases to consider on s: either $C = no\_match$, or E and B are empty. In the former case, $\mathcal{A}(s)$ is the state in which $E = no\_match$, and in the latter E is in solved form; in both cases, all guards of 4-4 are false.

Lemma 4 *Specification 4-6 is a refinement of specification 4-5.*

Lemma 4 is the final piece of the correctness argument for the transition-axiom specifications. This proof is more of a conventional refinement argument than the others, because the *consistency* actions of specification 4-5 are replaced by a sequence of actions of specification 4-6. In particular, each abstract *consistency* action corresponds to one concrete (lower-level) *consistency* action plus a number of *declash_I* actions. In proofs of this type of refinement, where one action is being replaced by a sequence, the trick is to define an abstraction function that maps exactly one low-level action in each sequence to the appropriate high-level action, and all others to stuttering actions [Lam83]. The complication comes from the possibility of interleaving of low-level actions from different high-level actions, representing overlaps in the execution of high-level actions. It is not alway possible to define an abstraction function on indiviual states, since the abstract value may depend on the history (or even future [AL88]) of the execution. To make the abstraction function definable, auxiliary information is added to the specification state to record an execution's history [OG76]. The auxiliary information does not change the specification; neither the transition axioms nor the liveness property in the augmented specification depend on auxiliary values.

In specification 4-6, each *consistency* action of specification 4-5 is replaced by a *consistency* action that does not check for term equality, and a sequence of *declash_I* actions. When the last *declash_I* action occurs, the abstract *consistency* action takes effect. In this case we to use history variables. With each equation e in the I component of specification 4-6, we

consistency
$((x \doteq t_1) \in \mathrm{B})$ & $(\mathrm{C} \neq \mathrm{no\_match}) \Rightarrow$
if $(\exists\ t_2 \mid (x \doteq t_2) \in \mathrm{S})$ **then**
$(\mathrm{B} := \mathrm{B} - (x \doteq t_1)$ & $\mathrm{I} := \mathrm{I} + \langle (t_1 \doteq t_2), (x \doteq t_1) \rangle)$
else $(\mathrm{B} := \mathrm{B} - (x \doteq t_1)$ & $\mathrm{S} := \mathrm{S} + (x \doteq t_1))$

declash_I
$(\langle (p \doteq t), (x \doteq s) \rangle \in \mathrm{I})$ & $(\mathrm{C} \neq \mathrm{no\_match}) \Rightarrow$
if $(p.\mathrm{head} = t.\mathrm{head})$
then $\mathrm{I} := \mathrm{I} + \langle (p[1] \doteq t[1]), (x \doteq s) \rangle + ... + \langle (p[n] \doteq t[n]), (x \doteq s) \rangle$
$- \langle (p \doteq t), (x \doteq s) \rangle$
else $\mathrm{C} := \mathrm{no\_match}$

Figure 4-13: Augmented axioms for specification 4-6.

associate the (binding) equation from B that caused e to be added to I. The elements in I are now pairs of equations $\langle i, b \rangle$, where i is an identity and b is the associated binding. Specification 4-6 is augmented by changing the type of I, and replacing axioms *consistency* and *declash_I* with the versions shown in Figure 4-13. Note that the augmented values do not affect allowable executions of the specification. A live execution of the augmented specification is also a live execution of the unaugmented one if the bindings in I are erased, and the converse is true because bindings can be added to translate to any live execution of the unaugmented specification.

As before, we define an abstraction function, $\mathcal{A}$, from specification 4-6 to specification 4-5, but in this case $\mathcal{A}$ is defined on states of the augmented specification in Figure 4-13.

$$\mathcal{A} : (\mathrm{container[equation]})^3 \times \mathrm{container}[\langle \mathrm{equation}, \mathrm{equation} \rangle] \times \{\mathrm{running}, \mathrm{no\_match}\} \rightarrow$$
$$(\mathrm{container[equation]})^3 \times \{\mathrm{running}, \mathrm{no\_match}\}$$
$$\mathcal{A}(\mathrm{E}_1, \mathrm{B}_1, \mathrm{S}_1, \mathrm{I}_1, \mathrm{C}_1) \equiv (\mathrm{E}_2, \mathrm{B}_2, \mathrm{S}_2, \mathrm{C}_2)$$
$$\text{where } \mathrm{E}_2 = \mathrm{E}_1 \ \&\ \ \mathrm{S}_2 = \mathrm{S}_1 \ \&\ \ \mathrm{C}_2 = \mathrm{C}_1 \ \&$$
$$\mathrm{B}_2 = \mathrm{B}_1 \cup \{b \mid (\exists i, \langle i, b \rangle \in \mathrm{I}_1)\}$$

(In the definition above, the container denoted $\{b \mid (\exists i, \langle i, b \rangle \in I_1)\}$ contains only one instance of each binding object b. There may be multiple occurrences of the same b in elements of I.)

The safety property for specification 4-6 again requires proving an invariant, $\mathcal{I}$, on the states, in this case states of the augmented specification. The invariant for specification 4-6 is the same

as for 4-5, but with additional constraints on equations in I.

$$\begin{aligned}\mathcal{I}(\langle \mathrm{E}, \mathrm{B}, \mathrm{S}, \mathrm{I}\rangle) \equiv\ & (e \in \mathrm{E} \Rightarrow \mathrm{is\_ground}(e.\mathrm{right})) \ \&\\ & (e \in \mathrm{B} \Rightarrow (\mathrm{is\_var}(e.\mathrm{left}) \ \&\ \mathrm{is\_ground}(e.\mathrm{right}))) \ \&\\ & (\mathrm{is\_solved\_form}(\mathrm{S})) \ \&\\ & (\langle i,\ b\rangle \in \mathrm{I} \Rightarrow (\mathrm{is\_ground}(i.\mathrm{left}) \ \&\ \mathrm{is\_ground}(i.\mathrm{right}) \ \&\\ & \quad \mathrm{is\_var}(b.\mathrm{left}) \ \&\ \mathrm{is\_ground}(b.\mathrm{right})))\end{aligned}$$

In the initial state, I is empty, so the abstraction function maps each of the other state components to the component of the same name in specification 4-5, and the abstract state is therefore also an initial one. Moreover, the invariant holds on the initial state. Next, show that if $a = (s_1, s_2)$ is a legal action of the augmented version of specification 4-6 such that $\mathcal{I}(s_1)$ holds, then $\mathcal{I}(s_2)$ follows and $(\mathcal{A}(s_1), \mathcal{A}(s_2))$ is a legal action of specification 4-5. Let $s_1 = (E_1, B_1, S_1, I_1, C_1)$ and $s_2 = (E_2, B_2, S_2, I_2, C_2)$.

- If a is a *declash_E* action, then $\mathcal{A}(a)$ is a *declash* action. Since I is unaffected, $\mathcal{I}$ is obviously maintained.

- If a is a *consistency* action, then there is an equation e_1 in B_1 such that is not in B_2. By the invariant on B_1, the left-hand side of e is some variable, x, so there are two cases depending on whether or not x occurs in S_1. If not, then $\mathcal{A}(a)$ is a consistency action; S_2 is still in solved form, so $\mathcal{I}(s_2)$ is true. If so (x does occur in S), then there is another equation e_2 in S_1 with x as a left-hand side. Both e_1 and e_2 have ground terms on the right, so adding the pair $\langle (e_1.\mathit{right} \doteq e_2.\mathit{right}), e_2\rangle$ to I_2 does not invalidate $\mathcal{I}$. In this case, $\mathcal{A}(a)$ is a stuttering action.

- If a is a *declash_I* action, then let $\langle (p \doteq t), (x \doteq s)\rangle$ be the element of I_1 that instantiates the guard. There are two cases, depending on whether $p.\mathit{head} = t.\mathit{head}$. If not, then $C_2 = \mathit{no\_match}$ and $\mathcal{A}(a)$ is a *consistency* action with the equation $(x \doteq s)$ instantiating the guard in specification 4-5. If $p.\mathit{head} = t.\mathit{head}$, then the child equations of $(p \doteq t)$ are added to I, each with an occurrence of $(x \doteq s)$. Since p, t, and s are ground, and x is a variable, $\mathcal{I}$ is maintained. The value of $\mathcal{A}(a)$ (when $p.\mathit{head} = t.\mathit{head}$) is a stuttering action except in the following case: if the arity of the head symbols is zero, so there are

no children, and there are no other pairs in I having $(x \doteq s)$ as second field, then an equality check has been completed, so $\mathcal{A}(a)$ is a (non-clashing) *consistency* action.

The liveness property for specification 4-6 obviously implies that of specification 4-5, because both use universal weak fairness, and every guard of specification 4-5 is also a guard of 4-6.

Correctness of the Procedures

As with most parallel programs, it is difficult to convince oneself that the transition procedures perform as desired without a careful correctness argument. We have asserted that the procedures in Figure 4-8 are linearizable and non-stopping. From the assumption that lower level data types (queues, substitutions, accumulators, and memory cells) are linearizable, it follows that the operations on these appear to be indivisible. Furthermore, the locality property of linearizability implies that in the entire system, which is a composition of linearizable objects, the operations take effect in an order consistent with the program order. It is therefore sufficient to consider interleaving of the operations on the individual state components. Part of the correctness argument in this stage of design is to show linearizability at the level of the transition procedures.

The proof obligation has been reduced to reasoning about interleavings of operations, so we show that the implementation is a refinement of its specification. The specification is the set of transition axiom procedure specifications, subject to the requirements of linearizability, productivity, and the non-stopping property. The proof is most like that of Lemma 4, since we are replacing high-level actions with a sequence of low-level ones. The main difference between correctness arguments for transition procedures and transition-axiom specifications is that procedures have internal control structures, so it may be necessary to consider the program counter as part of the lower level state. Note that the correctness condition is on the entire set of transition procedures, and the proof cannot be done separately for each procedure. The correctness of each procedure depends on what other things may be going on concurrently, which in this case is assumed to be other instances of the transition procedures. The proof is like a linearizability proof of an abstract data type, with the entire program state as the single abstract object.

As before, the proof depends on maintaining an invariant, $\mathcal{I}$, on the concrete objects. It is

```
declash_E = procedure (E, B: eq_queue, C: ptr[int],                                  1
                       E_size, B_size: accum) signals (stutter)                      2
                                                % E_size ≥ size(E)                   3
                                                % B_size ≥ size(B)                   4
    if (C = no_match) then signal stutter                                            5
    e: eqn := dequeue(E)                        % E_size ≥ size(E) + 1               6
              except when empty: signal stutter                                      7
    if is_var(e.left) then                                                           8
        {accum$add(B_size, 1)                   % B_size ≥ size(B) + 1              9
        accum$add(E_size, −1)                   % E_size ≥ size(E)                  10
        enqueue(B, e)}                          % B_size ≥ size(B)                  11
    else if (head(e.left) = head(e.right)) then                                      12
        {accum$add(E_size, arity(e.left) − 1)% E_size ≥ size(E) + arity(e.left)      13
        for (i := 1) to arity(e.left)                                                14
          enqueue(E, (e.left[i] ≐ e.right[i]))} % E_size ≥ size(E)+arity(e.left)−i   15
                                                % E_size ≥ size(E)                  16
    else C := no_match                                                               17
end declash_E                                                                        18
```

Figure 4-14: *Declash_E* procedure with invariant annotations.

not surprising that the invariant involves the relationship between *size* objects and the actual container sizes. (Technically, the *accumulators* must be read to extract the value. We omit the *accum$read* operations to reduce clutter.)

$$\begin{aligned}\mathcal{I}(\langle \text{E}, \text{B}, \text{S}, \text{I}, \text{C}, \text{E\_size}, \text{B\_size}, \text{I\_size}\rangle) \equiv\ & \text{E\_size} \geq \text{size(E)}\ \&\\ & \text{B\_size} \geq \text{size(B)}\ \&\\ & \text{I\_size} \geq \text{size(I)}\end{aligned}$$

Proving this invariant is straightforward by examining the actions within each procedure. Consider the *declash_E* procedures as an example. Roughly, any insertions to a container are preceded by an addition to the corresponding accumulator, while removals are followed by a subtraction. The code in Figure 4-14 contains annotations to show how operations within the procedure affect the relevant part of the invariant. Each comment is a predicate that holds after the statement on the same line has been executed. At the beginning of the procedure, the invariant is assumed, and any path through the code results in the invariant being maintained. (The predicate following the *dequeue* assumes an equation has been removed. If, instead,

the signal is raised, then *dequeue* raises the *stutter* signal and the values in the invariant are unchanged.) Similar arguments can be made for each of the other procedures.

Because the accumulator objects are upper bounds on the sizes of the respective containers, a zero accumulator value implies an empty container. When all three size objects are zero, then the three corresponding containers are empty, the transition axioms guards are all false, and a matching substitution has been computed. Thus, when the *termination* procedure observes a state in which all three size objects are zero, it should return *true*. However, correctness of the *termination* procedure does not follow from this fact alone, because the expression

$$((accum\$read(E\_size) = 0)\ \&\ (accum\$read(B\_size) = 0)\ \&\ (accum\$read(I\_size) = 0))$$

is not a single action—each of the *read* operations is a separate action. When the expression is executed, three different states may be observed, so it is not obvious that when the expression evaluates to true, there is a single state in which all three accumulators are zero. The correctness of *termination* depends on the order in which *reads*are done; it also depends on a global property of the system, that equations move in one direction through the containers. We assume that the programming language mandates left-to-right evaluation, which is true in C.

In any execution of the transition procedures, equations move from E to B and indirectly, to I, but no equation ever moves in the reverse direction, and no equation in B or I ever causes a new equation to be added to one of the preceding containers. Therefore, once E_size becomes zero, it will remain so. A predicate that remains true, once it becomes true, is said to be *stable*. Neither of the predicates $B\_size = 0$ or $I\_size = 0$ is stable, but the following three predicates are stable:

- $\mathcal{S}_1(s)$: $(s.E\_size\ = 0)$
- $\mathcal{S}_2(s)$: $((s.E\_size\ = 0)\ \&\ (s.B\_size = 0))$, and
- $\mathcal{S}_3(s)$: $((s.E\_size\ = 0)\ \&\ (s.B\_size = 0)\ \&\ (s.I\_size = 0))$.

(Each property is written as a predicate on some state s, and individual state components are accessed using the dot notation from records.) The stability of these implies that, if the test expression in *termination* evaluates to true, then all three accumulators are zero in the last state of the evaluation.

Stability of a predicate P is a safety property; it can be checked by showing that for any action $\langle s_1, s_2 \rangle$, if P is true in s_1, then P is also true in s_2. One often makes use of a previously proven invariant to show a stability property.

In proving the stability of $\mathcal{S}_1$, the only interesting procedure is *declash_E*, since none of the others modify *E_size*. (We refer to line numbers in the annotated version of Figure 4-14.) For each of the lower-level actions $\langle s_1, s_2 \rangle$ in *declash_E*, assume $\mathcal{S}_1(s_1)$ and show $\mathcal{S}_1(s_2)$. From $\mathcal{S}_1(s_1)$ and $\mathcal{I}(s_1)$, it follows that E is empty in s_1. There are only two actions that alter the value *E_size*, line 10 and line 13. In both cases, $\mathcal{I}(s_2)$ implies that in s_1, $E\_size \geq size(E)+1$, so $E\_size > 0$, and by contradiction, $E\_size = 0$ is not invalidated. We do not provide proofs of $\mathcal{S}_2$ and $\mathcal{S}_3$, which require the same type of reasoning.

As in the proof of Lemma 4, a proof of linearizability involves picking a single lower level action in any procedure that causes the abstract action. In Figures 4-15 and 4-16, the transition procedures are given with history variables added to show the values of the abstraction function at each point in the execution. The angle brackets are used to denote atomic operations, i.e., they group operations on history variables with some real operation in the program.

Non-stopping is obvious, because there is no recursion, and every loop is bounded by the arity of some operation, and there are no critical regions (at this level of abstraction).

It is interesting to note that the procedures are still correct if the operations on the *size* objects are done *after* the operations on the corresponding containers, but the proof becomes much more complicated. In particular, $\mathcal{I}$ is no longer an invariant, since child equations are added (to either E or I) before their size objects are incremented.

4.4 Discussion

In this section we summarize the main advantages of the transition-based approach, discuss its limitations and possible extensions, and then survey some of the related work.

4.4.1 Summary

One of our goals in developing the transition-based approach was to separate concerns of performance from those of correctness. It is instructive to see how the two concerns are addressed

```
declash_E = procedure (E, B: eq_queue, C: ptr[int],
                       E_size, B_size: accum) signals (stutter)
    if (C = no_match) then
        signal stutter                                              E1
    e: eqn := dequeue(E)
              except when empty: signal stutter
    if is_var(e.left) then
        {accum$add(B_size, 1)
        ⟨accum$add(E_size, −1)                                      E2
        E_abs := E_abs − (p ≐ t)
        B_abs := B_abs + (p ≐ t)⟩
        enqueue(B, e)}
    else if (head(e.left) = head(e.right)) then
        {⟨accum$add(E_size, arity(e.left) − 1)                      E3
        E_abs := E_abs + (p[1] ≐ t[1]) + ... + (p[n] ≐ t[n]) − (p ≐ t)⟩
        for (i := 0) to arity(e.left) enqueue(E, (e.left[i] ≐ e.right[i]))}
    else ⟨C := no_match                                             E4
        C_abs := no_match⟩
end declash_E

consistency = procedure (B, I: eq_queue, S: substitution, C: ptr[int],
                         B_size, I_size: accum) signals (stutter)
    if (C = no_match) then
        signal stutter                                              C1
    e: eqn := dequeue(B)
          except when empty: signal stutter
    assign(S, e.left, e.right)
        except when already_bound(t):
                {accum$add(I_size, 1)
                ⟨accum$add(B_size, −1)                              C2
                B_abs := B_abs − (p ≐ t)
                I_abs := I_abs + (t1 ≐ t2)⟩
                enqueue(I, (e.right ≐ t))
                return}
    ⟨accum$add(B_size, −1)                                          C3
    B_abs := B_abs − (p ≐ t)
    S_abs := S_abs + (x ≐ t1)⟩
end consistency
```

Figure 4-15: Annotated transition procedures–*declash_E* and *consistency*

```
declash_I = procedure (I: eq_queue, C: ptr[int], I_size: accum) signals (stutter)
        if (C = no_match) then
        signal stutter                                                          I1
        e: eqn := dequeue(I)
                except when empty: signal stutter
        if (head(e.left) = head(e.right)) then
            {⟨accum$add(I_size, arity(e.left) − 1)                              I2
            I_abs := I_abs + (p[1] ≐ t[1]) + ... + (p[n] ≐ t[n]) − (p ≐ t)⟩
            for (i := 0) to arity(e.left) enqueue(I, (e.left[i] ≐ e.right[i]))}
        else ⟨C := no_match                                                     I3
            C_abs := no_match⟩
  end declash_I

termination = procedure (C: ptr[int], E_size, B_size, I_size: accum) returns (bool)
        if (C = no_match) then
            return(true)                                                        T1
        if ((accum$read(E_size) = 0) &                                          T2
            (accum$read(B_size) = 0) &                                          T3
            (accum$read(I_size) = 0))                                           T4
            then return(true)
            else return(false)
  end termination
```

Figure 4-16: Annotated transition procedures–*declash_I* and *termination*

in various steps of the approach. The major correctness considerations are:

- Each of the transition-axiom specifications must solve the desired problem. This is proved by a refinement argument between specifications. The intent is to capture the complexity of the program-level parallel algorithm in these specifications, which are nondeterministic but still sequential.

- The transition axiom procedures must be effectively linearizable, assuming that the lower level abstractions are linearizable. If there is an interference relation on the lower level abstractions, it must be respected.

- The lower level abstractions must be linearizable, perhaps modulo an interference relation. This can be shown independently for each abstraction.

- The scheduler must meet the liveness property of the refined transition-axiom specification. This argument is often trivial, because the liveness property is weak.

Performance issues arise in a number of places as well.

- The transition-axiom specification used for the implementation must have a certain form so that it can lead to an efficient implementation.

- The transition-axiom procedures must be highly concurrent. This can be partially tested by running subsets of the procedures on carefully chosen data.

- The lower level objects must be highly concurrent, to the extent that they limit performance of the transition-axiom procedures.

- The scheduler must make effective use of the available processors. The ability to do performance tuning late in the program development process helps the programmer meet this goal.

A general conclusion is that thinking about nondeterministic transitions during program design is a good way to uncover program level parallelism. While transition axioms have been used to reason about parallel and distributed programs [CM88, LT87, Lam89], we know of no other work that defines a link between the transition axiom specification and a parallel program written in a conventional language.

4.4.2 Extensions

There are two aspects of the transition-based approach that limit its use. First, all parallelism is at a single level of abstraction; although there is concurrency between the transition axioms, each one is sequential. For small multiprocessors this is not only reasonable, but probably advantageous, because it significantly simplifies the scheduling problem. This simplification makes it easier to keep all processors busy, without the cost of supporting more threads than processors.

For larger machines, it would negatively affect modularity to require that all parallelism be at a single level of abstraction. It will be desirable to nest parallel procedures within each other. Extending the transition-based approach to meet these demands does not affect the transition axioms or the transition procedure, but it does affect the scheduler. To keep the same distinction between the transitions, which define legal actions, and the scheduler of those actions, a multi-level scheduler would be needed. The challenge would be to find a small but general set of mechanism for building multi-level schedulers, so that schedulers can still make effective use of processors.

Both of the examples in this thesis are programs that run in a batch style. Using the approach for reactive or interactive programs is possible, and if we assume the environment (or user) is infinitely fast at responding, then the same scheduling and synchronization strategies should work. Realistically, though, a program that interacts with the environment may be slowed by the rate of user input or other environmental responses. This does not affect the correctness of the program, but will affect performance.

4.4.3 Related Models and Methods

Many approaches to writing parallel programs have been proposed. Among those proposed for asynchronous (MIMD) machines, we divide the approaches according to whether or not they allow the full power of a conventional imperative programming style. An orthogonal distinction can be made between implicit and explicit parallelism.

For asynchronous machines, there are number of proposed programming paradigms that restrict the programming model. The most restricted is a pure functional language, in which no side-effects are allowed, so the complexity of nondeterminism from parallel evaluation is

avoided. However, in a functional language it is difficult to write programs that involve state changes, since this involves copying all or part of the state. For some programs, object mutations are mainly instantiations, i.e., an object may be created in some partially constructed form, and parallel threads fill in the missing pieces. This form of mutation can be handled by futures [Hal85], I-structures [ANP87], and logical variables in logic programs [Rin89]. Since objects cannot be mutated after instantiation, the only kind of race condition arises from whether or not an object has been computed. Special hardware or software checks can prevent threads from observing uninstantiated objects.

In any of these restricted paradigms, the possibility for implicit parallelism is greater than with imperative programs, because there is no chance of interference between expressions. In symbolic programs, however, where large complex objects are involved, the overhead of performing mutations by copying seems prohibitive, and there is no evidence to suggest that the advantage of parallelism can overcome this overhead. Although functional and related paradigms eliminate the need for explicit synchronization, there is still synchronization being done by the system. Consider, for example, the imperative program, $P_1(x); P_2(x); y := x$, where each of P_1 and P_2 mutate the object x. In a functional style, the procedures would be written to produce a new version of x, and the program would be written, $y = P_2(P_1(x))$. In theory, the evaluation of P_2 may begin while P_1 is computing, but in practice it may have to wait for P_1 to finish computing its result.

The FX programming language is a hybrid of the functional and imperative styles [Luc87a]. The type system distinguishes between purely functional expressions and mutating ones, so the compiler can automatically parallelize certain pieces of a program, while allowing the programmer the expressive power of assignment when it is necessary. Effect declarations in FX are analogous to our interference specifications, but in FX they are integrated into the type system, and must be verified by the compiler. This leads to a conservative approximation of correctness that is quite powerful in the absence of explicit synchronization, but does not reflect the abstract notion of correctness in general.

Implicit parallelism is also being used on imperative programs, although the compilation techniques must be more sophisticated than for more restricted programming models. The most significant effort has been in the area of parallelizing Fortran programs, where signifi-

cant progress has occurred in developing algorithms to detect parallelism for both numerical [ABC+87, PGH+89] and symbolic [LH88] programs. These techniques are most promising for numerical programs, most of the techniques focus on parallelizing loops. The payoffs for these compilation techniques on complete application programs running on existing machines are not yet known. All of the implicit techniques are conservative, and they typically do not generate the kind of parallelism in which concurrent tasks mutate data concurrently.

Message-passing paradigms, as exemplified by [CD90, Ame87], are popular for distributed memory machines. In the simplest of these, a program is organized around objects, and communication is done by sending messages to objects. Computation at a single object is sequential, so the programmer is left with the options of refraining from abstraction, i.e., not building any large objects, or of producing programs with little parallelism. In [CD90], objects can be grouped together into a multi-object abstraction. Our approach could be used with a message-passing language, although some of the low-level implementations would be different.

The Linda programming language [CG89] gives a uniform treatment to scheduling issues and synchronization, but includes a global space of dynamically created objects as part of the programming model. The scheduling strategies we use in the transition-based approach use a similar idea, i.e., tasks are placed in a shared heap. However, we do not mix synchronization into the same structure, since we separate synchronization from scheduling, viewing synchronization as a low level concern and scheduling as a high level one.

All of these approaches attempt to simplify parallel programming without giving up too much expressive power, but do not directly address issues of abstraction and correctness. Furthermore, none of these approaches offers a design method analogous to the transition-based approach. They present programming models, sometimes with examples to demonstrate expressive power, but do not give the programmer general guidance in the design process.

Chapter 5

Parallel Completion

In this chapter we describe a parallel solution to the completion problem for term rewriting systems. We present this example for two reasons. First, it demonstrates the utility of our transition-based approach. Second, completion is an important problem in term rewriting systems research, and this is the first parallel solution, either designed or implemented, for the problem.

The following properties of completion make it a good test case for the transition-based approach.

- There is no obviously efficient parallel solution. The first parallel solution we implemented was a straightforward parallelization of a well-known sequential solution, the Knuth-Bendix procedure. Not only is the performance of that implementation poor, but some of the performance problems are attributable to its similarity to the sequential solution. We believe these performance problems will appear in other applications as well, and that they are fundamental to the approach of developing parallel programs from sequential ones. We discuss this point further in Section 5.1.

- In principle, the basic steps in completion do not have to be performed in a particular order, and in many cases the steps are independent, so the problem does not appear to be inherently sequential. Moreover, unlike matching, applications of completion may run for minutes or even hours on realistic input, so there is enough computation to support large scale parallelism.

- Even in the sequential solutions, the order in which steps are taken plays a crucial role in performance. Some execution orders that are allowed in principle will generate enough intermediate values to exceed the memory capacity of most computers and the patience of most users. Thus, while completion has enough independent computation to support parallelism, it is not trivially parallelizable.

- The parallel solution produced using the transition-based approach is large: it is about ten thousand source code lines, has nine transition procedures, and thirteen different data types, of which five are concurrent. This makes modularity a real concern.

- Completion is typical of symbolic applications. It has large data structures that are, by necessity, dynamically allocated. It exhibits performance instability in the extreme. In the best case, an input is already in its final form, i.e., it is *complete*; the completion procedure simply tests for this property and returns the input. In the worst case, there is no finite complete solution, so the procedure generates an infinite sequence of successively closer approximations to the infinite solution.

Completion procedures have been used for doing data type induction [Mus80], interpreting equational logic programs [GM86], proving theorems in first order theories [HD83], debugging specifications [GGH90], proving equivalence of algebras [Mar86], and automatically generating equational unification algorithms [Hul80]. The original completion procedure was discovered by Knuth and Bendix [KB70], and has since been studied, modified, and extended. See [Buc85], for a historical survey of completion procedures, with more than 200 references that include algorithms, applications, and implementations.

A careful statement of the completion problem requires term rewriting theory that is unrelated to the problems of parallel program development. We therefore separate the details of completion from the features of our parallel implementation. In Section 5.1 we describe a sequential completion procedure in the abstract, and discuss some of the properties that make it a challenging procedure to parallelize. In section 5.2 we define the completion problem, and in Section 5.3 we present a transition-axiom specification for completion that is adapted from the presentation of [BDH86]. In Section 5.4 contains a directly implementable transition-axiom specification, from which the implementation is built, and in Section 5.5 we present some

```
complete = procedure (R : set of rewrite rules)
        do forever
            % inter-normalize:
                rewrite all rules (left and right sides) using all other rules
                    until nothing can be rewritten
                delete trivial rules

            % compute critical pairs:
                pick two rules and add their critical pairs to the rules
        end complete
```

Figure 5-1: Outline of a sequential completion procedure

performance results.

5.1 Opportunities for Parallelism

The completion process is rich with both opportunities and pitfalls for parallelism. Consider the outline of a sequential completion procedure given in Figure 5-1. The procedure manipulates a set of rewrite rules, each of which is a pair of terms as defined in Chapter 4.[1] It has two alternating phases: *inter-normalization* rewrites and eliminates rules, and *critical pairing* adds new rules.

5.1.1 Program Level Parallelism

There are a number of ways in which parallelism can be exploited at this level of abstraction, and we will use all of these in our implementation.

1. Inter-normalize in parallel. Each task could take one rule, and sequentially normalize it with respect to the others. If a rule rewrites to a trivial rule (right and left sides equal), then the rule was redundant and can be deleted. In parallel, however, care must be taken to avoid eliminating the last instance of a redundant rule. For example, a rule can always be used to rewrite itself to a trivial rule, so the algorithm must prevent such self-rewriting. The problem is more complicated, however, because rules that are equivalent up to their

[1] In this section, we gloss over many details of completion, and in particular, no distinction is made between equations and rewrite rules.

variable names can be used to rewrite each other to trivial rules. If rules r_1 and r_2 are variable renamings of each other, then, in parallel, r_1 can be used to trivialize r_2 while r_2 is being used to trivialize r_1.[2]

2. Compute critical pairs in parallel. The critical pair calculation compares the left-hand side of one rule with a subterm of another rule. Each task could work on a different subterm in parallel. In addition, although the outline in Figure 5-1 shows the critical pairing phase acting on a single pair of rules, multiple critical pair computations can be done at once.

3. Do the two phases, normalization and critical pairing, in parallel. In this case the procedure would be a two stage pipeline, with the set of rules as data, and a feedback loop from the second stage back to the first.

Before developing a solution using the transition-based approach, we considered parallelizing the sequential completion procedure using these three strategies. In our first attempt to parallelize completion, we used the third strategy, with the intent of incorporating the first two strategies later to balance the pipeline. Two lessons came out of that implementation effort, and lead us to abandon the approach of developing parallel programs from conventional sequential ones.

The first lesson, which is specific to completion, is that the amount of time spent in normalization far exceeds the time spent in critical pairing. The basic fact was known from sequential implementations, but the degree of difference—a factor of 20 was not unusual—was a surprise. The implication is that the potential speedup of the pipeline was only 5%. On the machine we were using, a six processor Firefly, dedicating one sixth of its processing power to the critical pairing stage is not a good use of resources.

The second lesson is of more general interest. Even if the target machine has a large number of processors, performance instability of the two tasks makes it difficult to balance the pipeline. The ratio of work (between the two stages) varies significantly across iterations of the outer loop. Figure 5-2 shows the relative time for the two stages of the pipeline, given typical input.

[2]In parallelizing Buchberger's algorithm for computing Gröbner bases, which is similar to the completion problem, Ponder notes the same phenomenon [Pon88].

norm (ms):	0	369	0	160	139	21	481	2781	170	922	1841	2897	4768
crit (ms):	28	0	19	32	15	47	151	35	48	94	117	180	67
norm/crit:	0	∞	0	5	9	0	3	79	3	9	15	16	71

Figure 5-2: Time spent in iterations of pipelined completion

Each column is a separate iteration, and the first two rows are the executions times for two two stages, rounded to the nearest millisecond (sometimes to zero). The third row shows the ratio of the two times (norm/crit). Note only does the ratio varies significantly, but the more expensive instances of normalization trail the emore expensive instances of critical pairing by one iteration. The explanation for this shadowing effect is that a more expensive instance of critical pairing generates many new rules, which produces a lot of work for the next iteration of normalization.

There are a number of way to address the specific performance problems of completion. For example, the critical pairing stage could work on more than one pair of rules to make that stage relatively more expensive, or perhaps inter-normalization could be optimized to make it relatively less expensive. In addition, if the stages were themselves parallelized, the shadowing effect could be handled by allocating processor resources dynamically, within each iteration. All of these proposals avoid the real problem: between the two stages is a synchronization point, when inter-normalization gets new rules from critical pairing, and critical pairing gets a normalized version of the rules from inter-normalization. This synchronization point is an artifact of having developed a parallel program that "looks like" the sequential one. Notice that if the first two strategies for parallelism had been used without the third, there would be two synchronization points per iteration, one after inter-normalization and one after critical pairing.

Inheriting unnecessary synchronization points is a general problem that is attributable to parallelizing sequential code, as opposed to writing parallel programs. In conventional sequential programming languages, the programmer is asked to give a total order on all statements, even though some partial order might be sufficient.

Dissatisfaction with the result of parallelizing the sequential completion procedure motivated the development of the transition-based approach described in Chapter 4. In this chapter we demonstrate its application to the completion problem. By describing transition axioms at the

appropriate level of abstraction, the approach allows parallelism both between critical pairing and inter-normalization and within them. Thus, all three of the parallelism suggestions outlined earlier are used in our implementation. The challenge in our implementation is to allow this level of concurrency without loosing track of what has been computed and what needs to be computed, i.e., without missing rewrite steps or critical pair calculations and without performing unnecessary work. To meet this challenge the parallel tasks must communicate frequently and inexpensively, which they do by reading and writing shared variables.

5.1.2 Lower Level Parallelism

The parallelism discussed in the previous section was all at the program level, but there is also parallelism to be found at lower levels of abstraction within completion. For example, rewriting a single term requires that all rules be applied to every subterm of the term. Parallel rewriting has been studied by others, including Dershowitz and Lindenstrauss [DL90], who presented an parallel implementation and then discuss some of the behavioral differences between parallel and sequential rewriting. In addition, parallel matching, which is used within rewriting, and parallel unification, which is used within critical pairing, have both been studied theoretically. The unification problem, has a linear time sequential algorithm, and is known to be P-Space complete, so it is unlikely to have a faster than polynomial time parallel algorithm [DKM84]. The matching problem has a logarithmic time algorithm on a polynomial number of processors [RR87, DKS88], but cannot be done in onstant time algorithm [VR90]. All of these results were shown using the PRAM model, which has limited applicability to real machines, because the model is synchronous, ignores communication overhead, and only bounds processor utilization within a polynomial of the input.

We chose to parallelize only at the level shown in Figure 5-1, rather than parallelizing within matching or unification. Given the performance results of Chapter 4, where only unusually large terms shows speedup from parallelism in matching, it is not appropriate to parallelize the matching problems that occur within completion. Similarly, the unification problems that occur within completion will typically not be large enough to warrant parallelization. The relative costs associated with parallelizing within rewriting (of a complete term) and parallelizing within critical pairing (of a single pair of rules) are less clear, but we did not parallelize either of those

operations because we found sufficient parallelism at a higher level.

5.2 Problem Statement

The definitions presented here are consistent with Bachmair, Dershowitz, and Hsiang [BDH86]. We assume a familiarity with the notions of terms, equations, and substitutions, as defined in Chapter 4, and we also continue the convention that, with possible subscripts, x is a variable, a and b are constants, f and g are arbitrary function symbols, and s and t are terms.

A term rewriting system R is a set of ordered equations called rewrite rules, written $s \rightarrow t$. A rewriting system R imposes a rewriting relation $\rightarrow_R$ on terms given by $s \rightarrow_R t$ if and only if:

- there is a rule $l \rightarrow r \in R$ and a matching substitution σ such that s contains σl as a subterm, and
- t is formed by replacing the occurrence of σl by σr.

For example, a system containing the rule $f(x) \rightarrow g(b)$ rewrites the term $g(f(a))$ to $g(g(b))$. Let $\rightarrow_R+$, $\rightarrow_R*$, and $\leftrightarrow_R*$ be, respectively, the transitive closure, the reflexive transitive closure, and the reflexive symmetric transitive closure of $\rightarrow_R$. Since $\leftrightarrow_R*$ is symmetric, it is well-defined even when the set of rules R is replaced by a set of (unordered) equations E. The relation $\leftrightarrow_E*$ is exactly the relation defined by the *equational theory presented by* E, also denoted $E*$. I.e., an equation $(s \doteq t)$ is true in $E*$ if and only if $s \leftrightarrow_E *$t.[3]

A rewriting system R is said to be *confluent* if and only if $r \rightarrow_R * s$ and $r \rightarrow_R * t$ implies there exists a term u such that $s \rightarrow_R * u$ and $t \rightarrow_R * u$. If $\rightarrow_R+$ contains no infinite chains, then R is said to be *noetherian.*[4] If R is both confluent and noetherian, it is said to be *convergent.* If R is convergent, then for any term t there exists a unique term s such that $t \rightarrow_R * s$ and s is irreducible (i.e., cannot be rewritten); in this case s is called the *normal form* of t in R, and is denoted $t \downarrow_R$.

Convergence implies that $s \leftrightarrow_R * t$ if and only if $s \downarrow_R = t \downarrow_R$, so the equational theory presented by a finite convergent R, with rules in R viewed as equations, can be decided by

[3] Recall from Chapter 4.3 that $(s \doteq t)$ denotes an equation.

[4] Another word for "noetherian" is "terminating." We use "noetherian" for the property on term rewriting systems, and "terminating" for the property on programs that manipulate term rewriting systems.

computing normal forms. When term rewriting techniques are applied to theorem proving [HD83], knowledge representation [Sch88], and logic programming [GM86], the property of convergence is often essential. In some applications, convergence is established by adding new rules to the system in a *completion* process, while in other applications convergence is a property that is tested for a fixed set of rules. Although the process of completion is not guaranteed to terminate, it is sometimes useful to perform completion for a fixed amount of time to compute an approximation to the infinite answer [GGH90]. All of these applications of term rewriting systems solve one of the following related problems.

Definition. Given a term rewriting system R, the *convergence decision problem* is to determine whether or not R is convergent.

Definition. Given a set of equations E, the *basic completion problem* is to find a convergent rewriting system R such that $\leftrightarrow_{R}*$ and $\leftrightarrow_{E}*$ are equivalent.

For some inputs E, no finite R exists to solve basic completion, so we generalize the problem as follows.

Definition. Given a set of equations E, the *completion problem* is to produce a (possibly infinite) sequence of rewriting systems $R_0, R_1, ...$ such that each R_i is noetherian, $\leftrightarrow_{R_i}*$ is contained in $\leftrightarrow_{E}*$, and for any equation $(s \doteq t)$ in $E*$ there is an i such that $s\downarrow_{R_j} = t\downarrow_{R_j}$ for all $j \geq i$.

If basic completion is solved for a given E, the resulting rewriting system can be used to decide the equational theory $E*$. Similarly, if completion is solved, the resulting sequence of systems can be used as a semi-decision procedure for $E*$. Basic completion is a special case of completion, and the convergence decision problem is solved as part of completion. Completion problems can be further generalized to allow new function symbols in the rewriting systems; for example, some extension of completion allow function symbols in R that are not in E, and require only that $\leftrightarrow_{R}*$ be a conservative extension of $\leftrightarrow_{E}*$. Other variations on the completion includes narrowing [Hul80] and completion modulo equations [Hue80, PS81]. Although these problems fall in the class addressed by our approach, we consider only traditional completion in this thesis.

All these variations on completion are unsolvable, and in fact the problem of determining whether a set of rules is noetherian is undecidable. Certain equations, like the commutative axiom cannot be ordered into a rewrite rule if we desire a noetherian system. The most common method for proving that a system is noetherian is to use a *reduction ordering* on terms, i.e., a monotonic well-founded ordering that is stable under substitution [Der82]. If for every rule in the system, the left side is greater than the right side by a reduction ordering, then the system is noetherian. We assume a reduction ordering is given as input to a completion problem, and permit a procedure that halts with failure if the ordering is not powerful enough to orient some equations that arise.

A completion procedure is specified in the following set of correctness conditions on the sequence of observed states.

Definition. A *completion procedure* takes a set of equations E and a reduction ordering $>$. It produces a possibly infinite sequence $R = \langle R_0, R_1, ... \rangle$ of rewriting systems such that:

1. For all R_i in R, and all $l \rightarrow r$ in R_i, $l > r$. (Each R_i is provably noetherian by the given ordering.)

2. For all R_i in R, $\leftrightarrow_{R_i}*$ is contained in $\leftrightarrow_E*$. (Each R_i is consistent with E.)

3. If there exists some R_i in R such that R_i is a solution to the basic completion problem, then the procedure halts with success, and the last element in R is a solution to basic complete. (The procedure terminates if possible.)

4. If no R_i in R solves basic completion, then either:

   (a) The procedure halts with failure.

   (b) R is infinite, and for any equation $s \leftrightarrow_E * t$ there is some R_i such that for all $R_j, j \geq i$, $s\downarrow_{R_j} = t\downarrow_{R_j}$. ($R$ is a solution to general completion.)

The first two conditions are safety properties, and the last two are liveness properties. One interesting aspect of these procedures is that they may run forever, but must continue to make progress towards finding a solution. This is ensured by condition (4b), which informally says that anything true in E must eventually be provable by rewriting. By condition (3), a procedure

is required to terminate if a solution to basic completion has been produced, the test for which involves deciding convergence. The problem here is simpler, however, because the test for a noetherian system is (conservatively) approximated by the given reduction ordering. As we will see, once R is known to be noetherian the test for confluence can easily be decided.

Unfortunately, by condition (4a), the specification admits trivial procedures that simply halt with failure on all inputs. Completion procedures typically differ on the set of inputs on which they fail, and the ability of a procedure to resist failure is one of the qualities by which procedures are compared. Completion procedures can be made *failure resistant* by allowing the reduction ordering to be modified in certain restricted ways during the completion process [DF85]. Completion procedures can be made *unfailing* by leaving some equations unordered and restricting the domain of terms to which rules can be applied [BDH86]. Because these generalizations complicate the completion process, we restrict our attention to completion procedures in which a fixed ordering ($>$) is given. Failure conditions will be discussed again after we describe the process through which new equations arise during completion.

5.3 A Transition-Axiom Specification

The transition axiom specification presented in this section is adapted from the description of *standard completion* by Bachmair, Dershowitz and Hsiang [BDH86]. It reformulates the original completion procedure of Knuth and Bendix [KB70] as a set of non-deterministically applied transition axioms.

The transition-axiom specification for completion relies on the following definitions. An *occurrence* is a location in a term and is denoted by a finite sequence of integers. The subterm of t at occurrence o, written $t|o$, is defined recursively: if o is the empty sequence, ϵ, then $t|o = t$, and if $o = \langle i, o_1, ...o_n\rangle$ and $t = f(t_1, ..., t_m)$, then $t|o = t_i|\langle o_1, ..., o_n\rangle$.

Let s and t be terms. If there exists a substitution σ such that $\sigma s = \sigma t$, then s and t are *unifiable*, and σ is their *unifier*. If σ is a unifier of s and t, and for all other unifiers σ', there exists a substitution τ such that $\sigma' = \sigma \circ \tau$, then σ is a *most general unifier* of s and t. A substitution τ is a *renaming* if for all variables v in the domain of τ, $\tau(v)$ is a variable. (Most general unifiers are unique up to variable renaming, i.e., up to composition with renaming substitutions. Furthermore, if any unifier exists for two terms, then a most general unifier

exists.)

Let r_1 be the rewrite rule $s \rightarrow t$ and r_2 be another rule $l \rightarrow r$, and assume r_1 and r_2 have no variables in common. Then $(s' \doteq t')$ is a *critical pair* of r_1 and r_2 if and only if:

- l is unifiable with a non-variable subterm of s at some occurrence o, with a most general unifier σ, i.e., $\sigma l = \sigma(s|o)$.
- s' is formed from σs by replacing $(\sigma s)|o$ by σr, and t' is σt.

Consider the following example. Let r_1 be the rule $g(f(x_1, y_1)) \rightarrow f(g(x_1), g(y_1))$, and let r_2 be the rule $f(f(x_2, y_2), z_2) \rightarrow f(x_2, f(y_2, z_2))$. In this case r_1 and r_2 have a critical pair defined as follows:

- Let o be $\langle 1 \rangle$, and σ be the substitution $\{f(x_2, y_2)/x_1, z_2/y_1\}$. Then σ is the most general unifier of $g(f(x_1, y_1))|o = f(x_1, y_1)$ and $f(f(x_2, y_2), z_2)$.
- Apply σ to r_1, yielding $g(f(f(x_2, y_2), z_2)) \rightarrow f(g(f(x_2, y_2)), g(z_2))$. Replace the left-hand side of this rule with $f(x_2, f(y_2, z_2))$ at occurrence o and turn the result into an equation. This produces the critical pair $(g(f(x_2, f(y_2, z_2))) \doteq f(g(f(x_2, y_2)), g(z_2)))$.

Given a pair of rules r_1, r_2, let $crit(r_1, r_2)$ denote the set of critical pairs of r_1 and r_2, with r_2's variables renamed if necessary, to avoid conflicting with the variables of r_1. Given a set of rewrite rules, R, let $crit\_all(R)$ denote the set of all critical pairs of rules in R, i.e., the union of all sets $crit(r_1, r_2)$ for r_1, r_2 in R. Note that r_1 and r_2 may be the same rule, so $crit\_all(R)$ contains $crit(r, r)$, for all r in R. Both *crit* and *crit_all* are unique up to variable renaming.

In addition to critical pair computations, a completion procedure performs inter-normalization during which rules are rewritten by each other and equations are rewritten by rules. When rewriting one rule by another, there is a technical problem when the two rules have left-hand sides that are renamings of each other, because either rule can be used to rewrite the other's left-hand side. The details of the problem are not important in this discussion, but the solution will affect our presentation. We associate an *age* with each rewrite rule and define the following predicate that will be used to restricted rewriting of the left-hand side of rules: if r_1 and r_2 are rewrite rules having left-hand sides that are renamings of each other, and r_1 is older than r_2, then r_1 and r_2 satisfy an *age restriction*, which we denote by the predicate $age\_restrict(r_1, r_2)$.

Using the age of rules to solve the technical problem is mentioned in the original inference rule formulation [BDH86], and the validity of this solution was confirmed by Dershowitz [Der90]. Associating an age with each rule will also solve the problem in the parallel implementation that was mentioned in Section 5.1 as the problem rewriting two rules to triviality by using each other; we will use rule age to keep one of the rewritings from taking place.

A transition-axiom specification for a completion procedure is given in Figure 5-3. It is similar to the inference rule description given in [BDH86]. The state consists of a container of equations E and a container of rewrite rules R. Initially, E holds the user's input and R is empty. (By convention, R and E are sets, since no duplicate rules or equations are inserted into either.) The notation ($s \ddot{=} t$), used in a number of the transition axioms, refers to the equation ($s \doteq t$) in either orientation. The value of the reduction ordering on terms is implicit in the use of $>$, and there is an invariant that all rewrite rules are *ordered* with respect to $>$, i.e., $l \rightarrow r$ implies $l > r$. We discuss each of the axioms in Figure 5-3 by informally describing the actions they defined. The discussion of axioms is followed by a definition the liveness property in Figure 5-3.

- *simplify:* Apply one rewrite step to either side of an equation.

- *delete:* Delete an equation with identical right and left hand sides, i.e., *a trivial equation*, from E.

- *orient:* Turn an equation into a rewrite rule using the input ordering ($>$) on the two terms.

- *right_reduce:* Apply one rewrite step to the right hand side of some rule. A rule $s \rightarrow t$ is right reduced by applying a rule r (possibly the same rule) to t, giving t'. An important property of reduction orderings is the following: if $s \rightarrow t$ and r are both ordered with respect to some reduction ordering $>$, then $s \rightarrow t'$ is also ordered with respect to $>$.

- *left_reduce:* Apply one rewrite step to the left hand side of a rule. In this case, the rule may become trivial, or it may have to be oriented in the reverse direction. Therefore, the rewritten rule is turned into an equation.

  The guard for *left_reduce* asserts that there are two rules in R, $s \rightarrow t$ and r, and that

State Components

E : container[equation] + ordering_failure
R : container[rule] + ordering_failure

Initially

E = user input
R = $\emptyset$

Transition Axioms

simplify

$(s \ddot{=} t) \in E$ & $(t \rightarrow_R t') \Rightarrow$
$E := E - (s \ddot{=} t) + (s \ddot{=} t')$

delete

$(s \doteq s) \in E \Rightarrow$
$E := E - (s \doteq s)$

orient

$(s \ddot{=} t) \in E$ & $s > t \Rightarrow$
$E := E - (s \ddot{=} t)$ & $R := R + (s \rightarrow t)$

right_reduce

$(s \rightarrow t) \in R$ & $(t \rightarrow_R t') \Rightarrow$
$R := R - (s \rightarrow t) + (s \rightarrow t')$

left_reduce

$(s \rightarrow t) \in R$ & $r \in R$ & $(s \rightarrow_{\{r\}} s')$ & $(\neg$age_restrict$(s \rightarrow t, r))$
$R := R - (s \rightarrow t)$ & $E := E + (s' \doteq t)$

deduce

$(s \doteq t) \in$ crit_all$(R) \Rightarrow$
$E := E + (s \doteq t)$

fail

$(s \doteq t) \in E$ & $(s \neq t)$ & $(s \downarrow_R = s)$ & $(t \downarrow_R = t)$ & $(s \not> t)$ & $(t \not> s) \Rightarrow$
$E, R :=$ ordering_failure

Liveness

CP fairness and CP termination

Figure 5-3: Transition-Axiom Specification for Standard Completion

s can be rewritten by r. The requirement $\neg age\_restrict(s \rightarrow t, r)$ prevents a rule from being used to rewrite its own left-hand side, and ensures that if one rule can be used to rewrite another, the second cannot be used to rewrite the first.

- *deduce:* Add one critical pair of R to E.
- *fail:* If there is a non-trivial equation in E that is in normal form with respect to R, and it cannot be ordered in either direction by $>$, then both E and R are set to *ordering_failure*. Such a state is called a *failed* state. No guards are true in a failed state.

Any procedure that performs a fair interleaving of these actions will solve the completion problem, although the required notion of fairness is rather technical:

Definition. An execution $(E_0, R_0), (E_1, R_1), ...$ is *CP fair* if and only if there exists some failed (E_i, R_i), which is necessarily the last state, or, for all indices i in the execution:

1. $(\cap_{j>i} E_j) = \emptyset$, and
2. if $e \in \cap_{j>i} crit\_all(R_j)$, then there exist some k and e' such that $e' \in E_k$ and e' is a renaming of e.

The first condition for CP fairness requires that any equation appearing in E is eventually ordered, simplified, or deleted. The second condition requires that every critical pair is eventually added to E. CP fair executions may be either finite, producing a convergent or failed system, or infinite, producing successive approximations to an infinite system.

To ensure termination we need a second liveness property.

Definition. An execution $(E_0, R_0), (E_1, R_1), ...$ is *CP terminating* if either it is finite or none of the R_i solve basic completion for E_0.

A completion procedure may fail if there is a non-trivial equation e in E that can be neither ordered or rewritten, since no fair execution can leave e in E forever.

5.4 A Refined Transition-Axiom Specification

The transition-axiom specification given in Figure 5-3 is not appropriate for direct implementation. Although we do not give a series of refined specifications as we did for matching, the same

issues must be addressed. In Section 4.3.2 we described three aspects of a transition-axiom specification that made it directly implementable: a weak liveness property, simple transition axioms (particularly the guards), and balanced granularity of the transition axioms.

Before presenting the directly implementable specification for completion, which is given in Figures 5-4 through 5-7, we summarize some of the ways in which the original specification is refined.

5.4.1 Weakening the Liveness Property

A significant portion of the design for parallel completion involves encoding the liveness property into state information, so that it is ensured by the safety property defined by the transition axioms. In the matching example, the liveness property in the original specification was encoded by separating the container of equations into a four different containers with different invariants on each. In the design for completion the same technique is used. The state components contain either equations or rewrite rules, and the invariants in this case are assertions about which equations and rules have been normalized with respect to some other rules, and which rules have had their critical pairs computed.

As with matching, the data structures contain information that would be part of the control structure of a sequential implementation. For example, in a sequential implementation of completion, one might have nested loops for rewriting all equations with respect to all rules; invariants about which equations are in normal form with respect to which rules depend on the control point (i.e., the value of the program counter) within each loop. In our parallel implementation, this control information is represented in the data structures. Similarly, to guarantee that all critical pairs are eventually added, rules that have had their critical pairs computed are stored in a state component having that property as an invariant. Our parallel implementations resembles a sequential transition rule implementation by Lescanne [Les], but our data structure are concurrent and have more control information to allow for the additional executions that come from parallelism.

Note that the liveness property does not require rules to be in normal form. However, practical experience from sequential implementations indicates that keeping rules inter-normalized is crucial to performance. Therefore, the program is designed with inter-normalization as a per-

formance requirement; information about the normalization relation between rules is encoded into the state.

Performance has to be considered even at this stage in the design. Normalization of equations, normalization of rules, and critical pair calculations are all operations that can be repeated multiple times on the same data without affecting program correctness. Thus, one straightforward approach to guaranteeing the liveness properties is to repeatedly normalize all equations and rules with respect to all rules, and compute all critical pairs between all existing pairs of rules. While this leads to technically correct executions, the cost in both time and space make it impractical. Again, this is well known from numerous sequential implementations. These performance considerations will result in the following design goals: an equation or rule should never be normalized multiple times by the same set of rules; the critical pairs of a given pair of rules should be computed at most once; only normalized rules should be used for critical pair calculations. These were our design goals, but there are cases in which our implementation may perform extra normalizations or work with rules that have not been normalized because outdated versions of objects are being used. This trade-off is made to keep communication overhead low.

5.4.2 Adjusting Minimum Granularity

The transition axioms in Figure 5-3 defined actions at the level of single rewrite steps and individual critical pair additions. Parallelizing at this level of granularity would incur considerable overhead, and is of no use on a small number of processors. Therefore, we adjust minimum granularity by combining, rather than splitting rules. This is the opposite process to the one performed for matching.

If too much rule combining is done, however, the resulting grain size be too large, and the problem of insufficient parallelism will surface. A rough complexity bound is placed on the computation required by each transition axiom, to achieve a performance balance between the transition procedures. The complexity measure used for the axioms in the refined specification is the following: assuming that operations on individual terms and rules require the a unit time to compute, each transition axiom requires no more than linear time, meaning linear in the total number of equations and rules in the system. Some transition axioms define actions that apply

State Components

NE : queue[equation]	% new equations
AE : queue[equation]	% all other equations
NO : queue[equation]	% normalized equations
NT : queue[equation]	% non-trivial equations
UO : queue[equation]	% unorderable equations
NR : queue[rule]	% new rules
SR : queue[rule]	% simplifying rules
LR : queue[rule]	% left reducers
RR : priority_queue[rule]	% right reducers
CR : queue[rule]	% critter
CD : queue[rule]	% critted
UC : queue[rule]	% uncritted
UR : queue[rule]	% unreduced rules
AR : queue[rule]	% all rules
H : {running, ordering_failure}	% halt flag

Initially

NE = user input

NO = AE = NT = UO = NR = SR = LR = RR = CR = CD = UC = AR = $\emptyset$

H = running

Figure 5-4: Directly Implementable Specification–Part I

a single rule, at most once, to (at most) every rule or equation in the system. Other transition axioms define actions that apply all rules as many times as possible to a single equation or rule.

An interesting aspect of this granularity adjustment is that it does not produce a refinement of the original specification in the classical sense of refinement. The actions of the refined specification are larger than the actions of the original specification, since each action of the refined specification is equivalent to some sequence of actions of the original specification.

5.4.3 Simplifying the Transition Axioms

A third type of refinement used in matching was the simplification of transition axiom guards. In particular, the guards in the directly implementable specification for matching had a simple form: they tested containers for the existence of an element, and they tested the value of scalar variables. Exactly the same style is used in the guards of the directly implementable specification for completion.

The guards in Figure 5-3 contain conditions on whether a term is in normal form, or whether an equation is orderable, or whether an equation is in the set of critical pairs of the entire system; all of these are costly to compute. The same technique that is used to weaken the liveness property is also used to simplify these guards: equations and rules are put into different state components, depending upon what invariants are true about them. The invariants on state components in the refined specification match the conditions in the guards of Figure 5-3, so a guard in the refined specification can simply depend on the existence of an item in a particular state component.

Figures 5-4 through 5-7 give a directly implementable specification for completion. For the reader's convenience, the state components specified in Figure 5-4 are also given in a graphical representation in Figure 5-5. Each of the boxes in Figure 5-5 represents a state component in the specification; the *queues* AE, AR, and UR share elements with other queues, but the rest are mutually disjoint. The "usual" path that data (i.e., rules and equations) take in Figure 5-5 is from top to bottom, with new equations being added to the topmost queue, NE.

The transition axioms make use of some subsidiary functions *left_reducible*, *right_reduced*, *rewrites*, and *right_reducible* that are defined below. In addition to listing the state components, Figure 5-4 gives their initial conditions. The state components have the following properties:

- NE is a *queue*[*equation*] that contains new equations about which nothing is known. They may be equations that have been input by the user (as in the initial state) or they may have been added as critical pairs of some rewrite rules. (None of the *queue*[*equation*]'s or *queue*[*rule*]'s in the specification need to be strict FIFO queues. A semi-queue, which has no total order on elements but ensures that anything enqueued will eventually be dequeued, would be sufficient. Our implementation uses FIFO queues.)

- AE is a *queue*[*equation*] that contains the union of NO, NT, and UO, which are the *queue*[*equation*]'s of normalized equations. When a new rewrite rule is added, this set of equations must be normalized with respect to that new rule.

- NO is a *queue*[*equation*] that contains normalized equations, where normalized means normalized with respect to all rules except the new ones (i.e., $AR - NR$, where the minus operation is subtraction on the set of elements).

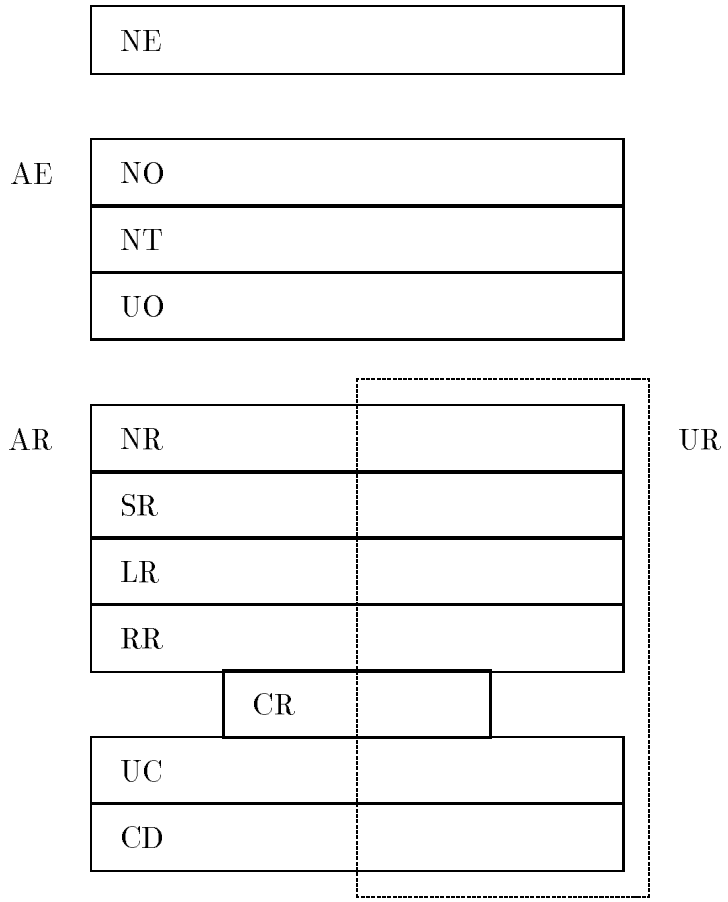

Figure 5-5: Venn diagram of state components for completion

Transition Axioms

normalize_eqn

$(e = \text{head}(\text{NE})) \Rightarrow$
$\text{NE} := \text{NE} - e$ &
$\text{NO} := \text{NO} + e \downarrow_{AR}$ & $\text{AE} := \text{AE} + e \downarrow_{AR}$

filter_eqn

$((s \doteq t) = \text{head}(\text{NO})) \Rightarrow$
$\text{NO} := \text{NO} - (s \doteq t)$ &
if $(s \neq t)$ **then** $\text{NT} := \text{NT} + (s \doteq t)$

orient_eqn

$((s \doteq t) = \text{head}(\text{NT})) \Rightarrow$
$\text{NT} := \text{NT} - (s \doteq t)$ &
if $(s > t)$ **then** $\text{NR} := \text{NR} + (s \rightarrow t)$ & $\text{AR} := \text{AR} + (s \rightarrow t)$
& $\text{AE} := \text{AE} - (s \rightarrow t)$
elseif $(t > s)$ **then** $\text{NR} := \text{NR} + (t \rightarrow s)$ & $\text{AE} := \text{AE} - (s \rightarrow t)$
& $\text{AR} := \text{AR} + (t \rightarrow s)$
else $\text{UO} := \text{UO} + (s \doteq t)$

back_simplify

$(r = \text{head}(\text{NR})) \Rightarrow$
$\text{NR} := \text{NR} - r$ &
$\text{SR} := \text{SR} + r$ &
$\text{NO} := \text{NO} - \text{rewrites}(\text{NO}, r)$ &
$\text{NT} := \text{NT} - \text{rewrites}(\text{NT}, r)$ &
$\text{UO} := \text{UO} - \text{rewrites}(\text{UO}, r)$ &
$\text{AE} := \text{AE} - \text{rewrites}(\text{AE}, r)$ &
$\text{NE} := \text{NE} + \text{rewrites}(\text{AE}, r)$

left_reduce

$(r = \text{head}(\text{SR})) \Rightarrow$
$\text{SR} := \text{SR} - r$ &
$\text{LR} := \text{LR} + r$ &
$\text{NR} := \text{NR} - \text{left\_reducible}(\text{NR}, r)$ &
$\text{SR} := \text{SR} - \text{left\_reducible}(\text{SR}, r)$ &
$\text{LR} := \text{LR} - \text{left\_reducible}(\text{LR}, r)$ &
$\text{RR} := \text{RR} - \text{left\_reducible}(\text{RR}, r)$ &
$\text{CR} := \text{CR} - \text{left\_reducible}(\text{CR}, r)$ &
$\text{UC} := \text{UC} - \text{left\_reducible}(\text{UC}, r)$ &
$\text{CD} := \text{CD} - \text{left\_reducible}(\text{CD}, r)$ &
$\text{UR} := \text{UR} - \text{left\_reducible}(\text{UR}, r)$ &
$\text{AR} := \text{AR} - \text{left\_reducible}(\text{AR}, r)$ &
$\text{NE} := \text{NE} + \text{left\_reduced}(\text{AR}, r)$

Figure 5-6: Directly Implementable Specification–Part II

right_reduce

$(r = \text{smallest}(\text{LR})) \Rightarrow$

$\text{LR} := \text{LR} - r$ &

$\text{RR} := \text{RR} + r$ &

$\text{NR} := \text{NR} - \text{right\_reducible}(\text{NR}, r) + \text{right\_reduced}(\text{NR}, r)$ &

$\text{SR} := \text{SR} - \text{right\_reducible}(\text{SR}, r) + \text{right\_reduced}(\text{SR}, r)$ &

$\text{LR} := \text{LR} - \text{right\_reducible}(\text{LR}, r) + \text{right\_reduced}(\text{LR}, r)$ &

$\text{RR} := \text{RR} - \text{right\_reducible}(\text{RR}, r) + \text{right\_reduced}(\text{RR}, r)$ &

$\text{CR} := \text{CR} - \text{right\_reducible}(\text{CR}, r) + \text{right\_reduced}(\text{CR}, r)$ &

$\text{CD} := \text{CD} - \text{right\_reducible}(\text{RR}, r) + \text{right\_reduced}(\text{CD}, r)$ &

$\text{UC} := \text{UC} - \text{right\_reducible}(\text{UC}, r) + \text{right\_reduced}(\text{UC}, r)$ &

$\text{UR} := \text{UR} + \text{right\_reduced}(\text{AR}, r)$

right_normalize

$((s \rightarrow t) = \text{head}(\text{UR})) \Rightarrow$

$\text{UR} := \text{UR} - (s \rightarrow t)$ &

if $(s \rightarrow t) \in \text{NR}$ **then** $\text{NR} := \text{NR} - (s \rightarrow t) + (s \rightarrow t \downarrow_{AR})$ &

if $(s \rightarrow t) \in \text{SR}$ **then** $\text{SR} := \text{SR} - (s \rightarrow t) + (s \rightarrow t \downarrow_{AR})$ &

if $(s \rightarrow t) \in \text{LR}$ **then** $\text{LR} := \text{LR} - (s \rightarrow t) + (s \rightarrow t \downarrow_{AR})$ &

if $(s \rightarrow t) \in \text{RR}$ **then** $\text{RR} := \text{RR} - (s \rightarrow t) + (s \rightarrow t \downarrow_{AR})$ &

if $(s \rightarrow t) \in \text{CR}$ **then** $\text{CR} := \text{CR} - (s \rightarrow t) + (s \rightarrow t \downarrow_{AR})$ &

if $(s \rightarrow t) \in \text{CD}$ **then** $\text{CD} := \text{CD} - (s \rightarrow t) + (s \rightarrow t \downarrow_{AR})$ &

if $(s \rightarrow t) \in \text{UC}$ **then** $\text{UC} := \text{UC} - (s \rightarrow t) + (s \rightarrow t \downarrow_{AR})$ &

if $(s \rightarrow t) \in \text{AR}$ **then** $\text{AR} := \text{AR} - (s \rightarrow t) + (s \rightarrow t \downarrow_{AR})$

add_critical

$(((r_1 = \text{head}(CR)) \;\&\; (r_2 = \text{head}(\text{UC}))) \mid (r_3 = \text{head}(\text{RR}))) \Rightarrow$

if $(r_1 = \text{head}(\text{CR})) \;\&\; (r_2 = \text{head}(\text{UC}))$ **then**

$(\text{UC} := \text{UC} - r_2)$ &

$(\text{CD} := \text{CD} + r_2)$ &

$(\text{NE} := \text{NE} \cup \text{crit}(r_1, r_2))$

else % Note: $r_3 = \text{head}(\text{RR})$

$(\text{UC} := \text{UC} \cup \text{CR} \cup \text{CD})$ &

$(\text{CD} := \emptyset)$ &

$(\text{CR} := \{r_3\})$ &

$(\text{RR} := \text{RR} - r_3)$ &

$(\text{NE} := \text{NE} \cup \text{crit}(r_3, r_3))$

fail

$(\text{UO} \neq \emptyset) \Rightarrow (\text{H} := \text{ordering\_failure})$

Liveness

Weak fairness between fail and the set of all other axioms.

Figure 5-7: Directly Implementable Specification–Part III

- NT is a *queue*[*equation*] that contains non-trivial, normalized equations. As with NO, the elements are normalized with respect to $AR - NR$. (Recall that non-trivial means that the right and left-hand sides are not identical.)

- UO is a *queue*[*equation*] that contains unorderable, non-trivial, normalized equations. Unorderable means that neither side of the equation is less than the other in the reduction ordering $>$. Again, normalized is with respect to $AR - NR$.

- AR is a *queue*[*rule*] that contains all the rules in the system. It has the same elements as $NR \cup SR \cup LR \cup RR \cup CR \cup UC \cup CD$, where $\cup$ is the union of queues considered as sets.

- UR is a *queue*[*rule*] that contains a subset of the elements in AR that are *not* right normalized. I.e., the right-hand side of all rules in $AR - UR$ are in normal form with respect to right reducers as defined below.

- NR is a *queue*[*rule*] that contains new rules about which little is known, except that the left-hand side is greater than the right-hand side by $>$. Since the ordering invariant is true of all rules in the system, we will not explicitly mention it for each of the state components.

- SR is a *queue*[*rule*] that contains *simplifying* rules, i.e., rules that are new, but have been used to normalize the equations in AE.

- LR is a *queue*[*rule*] that contains simplifying rules (as in SR) that *left reducers*, i.e., all other rules in AR have been left normalized with respect to LR.

- RR is a *priority_queue*[*rule*] that contains rules that are simplifiers, left reducers, and *right reducers*. A rule r is a right reducer if all rules in $AR - UR$ have been right normalized with respect to r. (Again, a semi-queue would technically be sufficient for the representation of RR. A *priority_queue*[*rule*] is a special of a semi-queue, assuming ordering on elements is well-founded. We use the relative size of rules as the ordering. A *priority_queue*[*rule*] is used in the implementation because rules dequeued from RR will be used to compute critical pairs, and sequential implementations had demonstrated the practical importance of computing critical pairs between small rules before larger ones.)

- CR is a single element $queue[rule]$ i.e., it is either empty or contains one rule. A rule in CR is called the *critter*. (We give CR the type $queue[rule]$, even though this is overly general, to avoid introduction of another data type in this presentation.) A rule r in CR has all the properties of a rule in RR, and in addition, the critical pairs for r and all rules in CD (below) have been computed. This invariant is complicated to state precisely, since a critical pair that has been computed and added to NE may already have been normalized, deleted, or oriented into a rewrite rule.

- UC is a $queue[rule]$ called the *uncritted* rules. All invariants for CR (and therefore CD) hold on rules in UC, and in addition the critical pairs of rules in UC and CD (below) have been computed. I.e., given a rule r_1 in UC and another rule r_2 in either UC or CD, $crit(r_1, r_2)$ have been computed and added to AE. UC is called the queue of *uncritted* rules because the critical pairs between the critter and the rules in UC have not yet been computed.

- CD is a $queue[rule]$ that contains the *critted* rules. I.e., it has all the properties of UC plus the additional properties that all critical pairs of rules in CD and the critter (CR) have been computed.

- H is used to record an ordering failure.

In the initial state, the set of new equations (NE) contains the user's input, and everything else is empty.

The transition axioms define actions that move rules from one queue to another, maintaining all the above invariants. They make use of the following predicates, where queues are viewed as sets.

- If Q is either a set of equations or a set of rules, then $rewrites(Q, r)$ is the subset of Q that can be rewritten using r.

- If Q is a set of rules, then $left\_reducible(Q, r)$ is the subset of Q that can be rewritten on the left-hand side by r.

- If Q is a set of rules, then $right\_reducible(Q, r)$ is the subset of Q that can be rewritten on the right-hand side by r.

- If Q is a set of rules, then *right_reduced*(Q, r) is the set *right_reducible*(Q, r) with each right-hand side rewritten once by r.

We informally describe the axioms in Figures 5-6 and 5-7 by describing the actions that each axiom defines. *Normalize_eqn* actions remove an equation from the NE, normalize it with respect to AR, and add it to NO. *Filter_eqn* actions remove a normalized equation from NO, check to see whether it is a trivial equation, and if not, insert it into NT. *Orient_eqn* actions remove an equation from NO and orient it, if possible, using the given reduction ordering. If neither orientation is consistent with the reduction ordering, then the equation is added to UO; otherwise the resulting rule is added to NR. When a new rule is added by an *orient_eqn* action, the equations that are in normal form must be rewritten by the new rule to ensure they are still in in normal form. This is done by a *back_simplify* action; any equation than can be rewritten by the new rule may no longer be in normal form with respect to other rules, so it is moved to NE. *Left_reduce* and *right_reduce* are similar to *back_simplify*, but in these cases other rules are rewritten by the new one. If a rule is left-reduced, it is moved back to NE, since it may have to be deleted or oriented in the other direction. If a rule is right-reduced then it is moved into UR, which marks it as not having a normalized right-hand side. The *right_normalize* actions take rule from UR and normalize them. Critical pair computations are done by *add_critical*, and the new equations are added to NE. *Fail* actions are used to stop the computation when an unorderable equation (which is also normalized and non-trivial) has been found.

The liveness property in Figure 5-7 is *weak fairness* between the *fail* actions and all others. In other words, if a *fail* action is continuously enabled, it must eventually be taken. Thus, an unorderable equation may exist in UO for a long time before the completion process fails, but unless the equations is moved or deleted, the process must eventually halt with failure. Note that there is no fairness requirement between any of the other axioms.

The transition axiom specification given in Figures 5-4 through 5-7 have been used as the design of a parallel completion procedure, which is discussed in the next section. The invariants stated above for each of the queues in the completion procedure state would be an important piece of a correctness proof to show that the refined specification satisfies the specification in Figure 5-3. Although the program has been tested on a number of interesting examples, we believe that rigorous correctness arguments, along the lines of those given for the matching

program in Chapter 4, would be an interesting exercise for completion program. However, such proofs are beyond the scope of this thesis.

5.5 Scheduling and Performance

In this section we describe some of the schedulers for completion and present performance numbers. To simplify this discussion, we consider only inputs on which completion produces a convergent set of rewrite rules without failing. Our implementation does not have a user interface that allows the program to be used as a semi-decision procedure, so we can only give performance numbers for executions that halt. In addition, since our implementation cannot be used as a semi-decision procedure, it only fails if there is no other transition procedure that can take a step, and since we do not include such examples, the scheduling of the *fail* procedure is not discussed here.

We begin by presenting the performance for our best scheduler, and then discuss some of the alternatives that we considered. There is an important difference between the numbers presented here and the numbers presented in Chapter 4: the inputs given for completion are realistic examples. Most of the examples are algebraic, and some are taken from the term rewriting literature.

These examples are executed on Firefly with 6 CVAX processors, varying the number of threads between 1 and 6. Although we intended to compare our parallel solution to a sequential solution that was implemented first, the parallel program (running on one processor) is significantly faster than the sequential program. In matching, where both the parallel and sequential programs were relatively simple, it was not difficult to ensure that the parallel and sequential programs were consistent. When appropriate, optimizations applied to the parallel matching program were also applied to the sequential matching program. This was much more difficult for completion, and we eventually abandoned the sequential program as a baseline. One indication that our implementation is reasonably fast in an absolute sense is a comparison to the implementation of completion in the Larch Prover [GGH90]. The Larch Prover implementation has been used for a number of large examples, and is consistently slower than our implementation running on one processor. Admittedly, the Larch Prover implementation is more powerful than ours, especially in its semi-automatic approach to proving termination, but

	1 abs	1 : 2	1 : 3	1 : 4	1 : 5	1 : 6
grp	1718	2.0	2.4	2.1	4.1	2.9
mult	4666	1.9	2.7	3.4	3.3	2.9
fib4	6603	1.8	3.1	2.9	2.2	3.5
grp2hom	11980	2.0	2.8	2.5	4.2	4.2
grp56	18796	2.2	2.3	3.1	3.7	3.8
domino	44162	2.0	2.9	3.7	3.8	5.1
dom	55871	1.9	2.9	3.7	4.2	4.8
domino2	55585	1.9	2.9	3.7	4.3	4.8

Figure 5-8: Transition-based completion using the best scheduler.

the comparison demonstrates that our implementation is fast enough to be of practical use.

Figure 5-8, gives performance results for 8 example inputs. The first column of numbers is the absolute performance of the program running on one processor, i.e., using a scheduler with only one thread. Each of the other columns gives the relative performance as the number of processors is increased. The numbers were obtained by averaging five executions of each example. Note that the first three examples are relatively short executions; typically the speedups are better with the larger examples.

The scheduler for completion is very similar to those for matching. The scheduler used to produce the numbers in Figure 5-8 executes the transition procedures in the order given by the specification in Figures 5-6 and 5-7. This scheduler repeatedly invokes the same transition procedure until it stutters, at which point it invokes the next procedure named in the specification. The only exception is the *add_critical* procedure, which execution only once before repeated the other procedures. In choosing this scheduler, we are using one of the lessons learned from sequential implementations of completion: performance, both in time and space, is better if equations and rules are kept in normal form. Thus, as soon as there is new inter-normalization work to be done, the critical pair computations are stopped.

A similar scheduling strategy to the one used for Figure 5-8 is to execute a single instance of *add_critical*, and then begin normalization again, even if no new equations were added. This seems to be a worse strategy, at least in the 1 processor case, because when no new equations have been added there should be nothing normalize. However, the performance of this scheduler was not noticeably different than when the scheduler waited for a new equation to be added

	1 abs	1 : 2	1 : 3	1 : 4	1 : 5	1 : 6
grp	3994	1.8	2.6	3.3	3.9	3.6
mult	4480	1.7	2.4	3.4	3.5	3.6
fib4	6699	1.9	2.7	2.0	2.5	3.2
grp2hom	11687	1.9	2.8	3.4	3.9	4.4
grp56	77037	7.8	7.0	9.8	11.5	15.1
domino	44609	2.0	2.0	3.7	4.4	4.9
dom	56740	1.9	2.8	3.1	4.2	3.0
domino2	57603	1.9	2.8	3.4	4.3	4.7

Figure 5-9: Transition-based completion using a round-robin scheduler.

before starting normalization.

Another scheduler that seems like a bad idea is a round-robin scheduler. In this case each transition procedure is executed exactly once before going on the next one. Since more than one critical pair may be added by a single *add_critical* invocation, rules and equations are sometimes not inter-reduced. The performance using this scheduler is shown in Figure 5-9. An interesting aspect of these numbers is that the *grp56* example gets super-linear speedup. The performance of this example is highly dependent on the order in which critical pairs are computed, because there is one critical pair that eliminates most of the other rules [Mar86]. That explains why the speedup is super-linear. The other interesting point about the *grp56* example is that while the speedup is much better for the round-robin scheduler than for the scheduler in Figure 5-8, the absolute performance is not. The main difference is that the round-robin scheduler is much slower on one processor, giving the illusion of great performance on 6 processors. The same is true of the *grp*, where the speedup improves from 3.2 to 3.6, but absolute performance is worse. The conclusion is that looking at speedups without considering absolute performance is a very bad way to tune parallel program performance.

The choice of scheduler can have a dramatic affect on performance. Another scheduler that we considered invokes each of the transition procedures, including *add_critical*, until the procedure stutters. In this case the state may become quite large while critical pairs are being computed. For most inputs, this scheduler ran out of space before finding an answer.

The main optimization used in the matching scheduler was to increase the granularity of transition procedures. This does not appear to be a problem for completion, and for small inputs

the performance appears to to limited more by lack of parallelism than by scheduling overhead. Increasing the granularity of the completion tasks would be much harder than with matching, because most of the parallelism is process parallelism rather than data parallelism. One might, for example, wish to combine the *filter* procedure with the *normalize_eqn* procedure, but this requires restructuring the state and significantly changes those two transition procedures.

The final two sets of performance numbers are not directly related to performance tuning of the application scheduler. Instead, they consider some of the effects of the underlying system scheduler. In the previous sets of the numbers, the threads of the application scheduler were *pinned* to particular processors, i.e., they are not moved from one processor to another by the operating system. As shown in Figure 5-10, which used the same application scheduler as Figure 5-8, the performance is significantly worse when threads are not pinned, because of the overhead of scheduling, particularly the loss of cache context.

	1 abs	1 : 2	1 : 3	1 : 4	1 : 5	1 : 6
grp	1703	1.9	1.9	2.1	2.3	2.2
mult	4693	1.8	2.5	2.8	3.0	2.9
fib4	7094	1.9	3.1	2.9	2.7	2.3
grp2hom	11753	1.5	2.1	2.8	3.2	2.6
grp56	17460	1.6	2.1	2.5	2.0	2.5
domino	45139	1.8	1.8	2.2	2.5	3.3
dom	56266	1.1	2.3	3.2	2.8	3.3
domino2	56507	1.8	2.0	3.3	3.0	2.8

Figure 5-10: Performance when threads are not pinned to processors.

The performance results for completion are encouraging. For the larger examples, performance continues to improve as each processor is added. This does not imply scalability beyond a small number of processors, but at least we found no evidence against scaling the completion process. In our implementation, some of the data structures have single synchronization points, so on a larger multiprocessor they would probably have to be changed to reduce contention or improve locality. Fortunately, the queues need not be strict FIFO queues, so an efficient distributed implementation of the queues should be possible.

Our procedure is significant from an algorithmic standpoint as well: the completion process is complicated and the parallel procedure differs in non-trivial ways from sequential ones.

We used the abstract description of completion given by Bachmair et al [BDH86] as a starting point for our design, but there is a large step between their description and the directly implementable transition-axiom specification that describes our procedure. In particular, the liveness requirement, CP fairness, was encoded into the data structures without creating too many serialization points.

Chapter 6

Summary and Conclusions

In this thesis we introduced a new approach to the design and implementation of parallel programs that is intended for applications characterized by irregular data and control structures. Programs with such irregularities have proved especially difficult for simpler programming models that rely on compiler detected parallelism, or use only strict data parallelism. Our work is therefore based on a programming model with explicit parallelism and mutable data.

The transition-based approach, presented in Chapter 4, addresses the problem of program synthesis by breaking the development process into four distinct phases; each phase has clearly stated correctness and performance requirements. The approach encourages the discovery of program level parallelism that includes both data and process parallelism. The emphasis is on high level concerns such as: finding the right task unit for parallelism, determining the kinds of shared objects that will be used in the state, and choosing a scheduling strategy for the tasks.

A program is implemented as a set of transition procedures that are implemented to have the behavior of indivisible operations, but the performance of highly concurrent operations. Chapters 2 and 3 dealt with how requirements can be met, focusing on the lower level concerns of how to build parallel program modules, and how to specify their interfaces. Chapter 2 gave a precise meaning to "indivisible," which is used not only for the transition procedures, but for subsidiary data abstractions on which those procedures are built. Chapter 3 gave examples of implementations of data abstractions that exhibit a high degree of concurrency.

Chapter 2 examined foundational issues relevant to what program modules should look like and how correct behavior should be defined. We examined existing correctness notions,

extended notions from multiprocessor memory models to general concurrent data types, and introduced a new notion that extends linearizability to allow for interference specifications. The key insight behind this extension is that in practice, many concurrent data types are not used with concurrency between all pairs of operations. Furthermore, this lack of concurrent invocation is useful information to the data type's implementor, since in many cases a simpler or more efficient implementation can be used. By adding explicit interference information to a data type's specification, this information becomes part of the contract between user and implementor. We also defined a liveness property called non-stopping that is appropriate for concurrent data types; non-stopping precludes, for example, implementations that either deadlock or loop.

The emphasis in Chapter 2 was on how to build clean abstractions, with secondary attention given to performance. The emphasis in Chapter 3 was on how to get good performance, assuming the requirements for abstraction from Chapter 2. Performance of a parallel program is an issue of latency—how long does the user have to wait? Performance of concurrent data types involves a combination of latency and throughput; one is sometimes willing to allow longer latency for individual operations, if it means that an object can handle multiple concurrent operations faster than the sum of their latencies. The programming technique employed in Chapter 3 involved fine-grained synchronization, where critical regions typically involved a small fixed number of instructions. Because the overhead of invoking a synchronization primitive was relatively high, we also used coherent shared memory without explicit synchronization. Two specific example implementations were presented: a queue implementation that grows dynamically, and a mapping implementation that yields high throughput on key assignments. We concluded from these examples that while highly concurrent mutable objects involve intricate coding, abstraction can often be used to localize the concurrency concerns.

We demonstrated our approach on two example programs, matching and completion. Both programs were designed using the transition-based approach, and the implementations done using highly concurrent data types that are linearizable modulo their interference specifications. The performance results in both cases provided evidence that the transition-based approach can be used to develop efficient programs.

As expected, the overhead of parallelism outweighed any benefit of parallelism on realistic

matching examples; however, the simplicity of the problem made it possible to analyze the effects of the transition-based approach on performance. We showed that for certain classes of input data, the application scheduler could be tuned to produce nearly linear speedups, although a scheduler tuned for one class often performed badly on another. This performance instability led us to choose a scheduler that was less than optimal on some of the easy cases, but handled the aberrant cases more gracefully. The conclusion was that to adequately address performance instability, one must pay the cost of allowing late binding on scheduling decisions.

The completion problem validated the transition-based approach as something useful, as opposed to just interesting. Our program performs well on realistic inputs, and improved with each addition of a processor. The completion procedure is a contribution, in itself, to the field of term rewriting research. That our procedure has an optimized implementation on a real multiprocessor proves the practical nature of our procedure. As with matching, parallelism does not pay off until there is sufficient computation to outweight the overhead of parallelism. With completion, however, there are many interesting examples for which the parallel implementation is faster than the sequential one. For the *group* axioms, a classic problem for completion procedures, parallelism proves to be beneficial, and on some larger examples the performance gain from six-fold parallelism is between four and five.

All of our experimental work was done on a six processor, shared-bus machine, although the approach was designed with a view toward larger machines and alternate models of memory. One indication of scalability in our designs is that even on the shared memory we used multi-ported objects, which contain thread-specific data within the object implementation. This provides an abstract model for concurrent objects that will allow us to move from centrally stored objects to distributed ones, without changing the object interfaces. We plan to pursue these ideas on distributed memory multiprocessors by develop a library of multi-ported objects for a variety of machines. The data types implemented as part of matching and completion will provide a starting point for this library, and porting these objects to new architectures would be a reasonable next step. A library of objects with consistent interfaces across machines, but highly tuned implementations on each, could provide the basis for portable and efficient application development.

One fundamental assumption in our implementations, although not in the high level designs,

is that the machine provides sequentially consistent memory. Modifying the code to remove this dependence would require the addition of explicit synchronization. Adding such synchronization conservatively would not be difficult, but the impact on performance could be quiet significant. In that regard the dependence on sequentially consistent memory may be useful: since the code is designed to perform well on sequentially consistent memory, it provides a benchmark for comparing the performance of weaker memory models. In addition, if the semantics of memory remains unchanged, the programs could be ported to machines with more processors or non-uniform memory access; in this case the issue becomes one of performance rather than correctness. We envision having to replace some of the shared objects with distributed versions to remove bottlenecks that might arise in motr loosely coupled architectures, but again, the current object interface allows for this change without affecting the higher level program.

There is no theoretical limit on the number of processors that can be used with the transition-based approach, but as discussed in Chapter 5, there is a pragmatic limit because a program produced by this method has only one parallel algorithm, the scheduler. To understand this limit, recall that data parallelism is obtained by allowing multiple instances of the same transition procedures to execute concurrently, while process parallelism involves different transition procedures executing concurrently. As with any method, process parallelism requires that different code be written for every distinct process; the limitation in our approach is that this code, which is the set of transition procedures, is implemented as a single abstraction. There is a limit to how many procedures one can reasonably expect to have in a single abstraction. To remove this limit, one needs the ability to nest parallel algorithms within others, which implies a multi-level scheduler. The hard part of such an extension is to engineer the interface between the application scheduler and the transition procedures. For our one-level scheduler, this interface is comprised of the scheduler requirements in Chapter 4, and the low level design decisions of Chapter 3. For a multi-level scheduler, the performance implications of these decisions would have to be rethought.

There are still many gaps in our understanding of how to build well-structured yet efficient parallel programs. However, many of the lessons learned here will apply to other architectures and different application domains, and we believe that the approach is a first step down a promising path.

Bibliography

[ABC+87] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the 1987 International Conference on Supercomputing*, 1987.

[ABLL90] Thomas Anderson, Brian Bershad, Edward Lazoswka, and Henry Levy. Scheduler activations: Kernel support for effective user-level thread management. Technical Report 90-04-02, University of Washington, Seattle, Washington, October 1990.

[AH90a] Sarita V. Adve and Mark D. Hill. Implementing sequential consistency in cache-based systems. In *Proceedings of the International Conference on Parallel Processing*, April 1990.

[AH90b] Sarita V. Adve and Mark D. Hill. Weak ordering–a new definition. In *17th International Symposium on Computer Architecture*, April 1990.

[AL88] Martín Abadi and Leslie Lamport. The existence of refinement mappings. Research Report 29, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, 1988.

[Ame87] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of the ECOOP*, pages 234–242. Springer-Verlag, June 1987.

[And89] Thomas Anderson. The performance implications of lock management alternatives for shared-memory multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*. IEEE, 1989.

[ANP87] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. Computation Structures Group Memo 269, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Massachusetts, February 1987.

[AW91] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. In *Symposium on Parallel Algorithms*. ACM SIGACT-SIGARCH, 1991.

[BDH86] Leo Bachmair, Nachum Dershowitz, and Jeih Hsiang. Orderings for equational proofs. In *Proceedings of the Symposium on Logic in Computer Science*, pages 346–357. IEEE, 1986.

[BGHL87] Andrew D. Birrell, John V. Guttag, James J. Horning, and Roy Levin. Synchronization primitives for multiprocessor: A formal specification. Technical Report 20, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, 1987.

[BHJ+87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in emerald. *IEEE Transactions on Software Engineering*, pages 65–76, January 1987.

[Bir89] Andrew D. Birrell. An introduction to programming with threads. Technical Report 35, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, 1989.

[BR90] Roberto Bisiani and Mosur Ravishankar. PLUS: A distributed shared-memory system. In *17th International Symposium on Computer Architecture*, pages 115–124, 1990.

[BT88] H. Bal and A. Tannenbaum. Distributed programming with shared data. In *International Conference on Computer Languages*, 1988.

[Buc85] B. Buchberger. Basic features and development of the critical pair completion procedure. In *Proceedings of the First International Conference on Rewriting Techniques and Applications, Dijon, France*, pages 1–45. Springer-Verlag, LNCS 202, May 1985.

[CD90] Andrew A. Chien and William J. Dally. Experience with concurrent aggregates (CA): Implementation and programming. In *Proceedings of the Fifth Distributed Memory Conference*, April 1990.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[CL88] Hubert Comon and Pierre Lescanne. Equational programs and disunification. Research Report 904, Unité de Recherche INRIA-Lorraine, Domaine de Voluceau Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France, September 1988.

[CM88] K. Mani Chandy and Jayadev Misra. *A Foundation of Parallel Program Design.* Addison-Wesley Publishing Company, 1988.

[Dal86] William J. Dally. *A VLSI Architecture for Concurrent Data Structures.* PhD thesis, California Institute of Technology, Pasadena, California, March 1986.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[Der90] Nachum Dershowitz, 1990. Private communication.

[DF85] Dave Detlefs and Randy Forgaard. A procedure for automatically proving termination of a set of rewrite rules. In *Proceedings of the First International Conference on Rewriting Techniques and Applications, Dijon, France*, pages 255–270. Springer-Verlag, LNCS 202, May 1985.

[Dij65] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[DKM84] Cynthia Dwork, Paris C. Kanellakis, and John C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1:35–50, June 1984.

[DKS88] Cynthia Dwork, Paris C. Kanellakis, and Larry Stockmeyer. Parallel algorithms for term matching. *SIAM Journal of Computing*, 17(4):711–731, August 1988.

[DL90] Nachum Dershowitz and Namoi Lindenstrauss. An abstract machine for concurrent term rewriting. In *Proceedings of the 2nd International Conference on Algebraic and Logic Programming*, Berlin, October 1990. LNCS, Springer.

[DS88] Michel Dubois and Christoph Scheurich. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, February 1988.

[Ell85] Carla Schlatter Ellis. Distributed data structures: A case study. *IEEE Transactions on Computers*, C-34(12):1178–1185, December 1985.

[GGH90] Steven J. Garland, John V. Guttag, and James L. Horning. Debugging larch shared language specifications. Technical Report 607, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, 1990.

[GHW85] John Guttag, James Horning, and Jeannette Wing. Larch in five easy pieces. Research Report 5, Digital Equipment Corporation Systems Research Center, July 1985.

[GJLS87] David Gifford, Pierre Jouvelot, John Lucassen, and Mark Sheldon. Fx-87 reference manual. Technical Report 407, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, MA, 1987.

[GL90] Kenneth Goldman and Nancy Lynch. Modelling shared state in a shared action model. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, June 1990.

[GLL+90] Kaourosh Gharachorloo, Daniel Lenoski, James Laudon, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th International Symposium on Computer Architecture*, pages 15–26, 1990.

[GLR83] Allan Gottlieb, B.D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processes. *ACM Transactions on Programming Languages and Systems*, pages 164–189, April 1983.

[GM86] Joseph A. Goguen and José Meseguer. EQLOG: Equality, types, and generic modules for logic programming. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming. Functions, Relations, and Equations*, pages 295–363. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Hal85] Robert Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, October 1985.

[HD83] Jieh Hsiang and Nachum Dershowitz. Rewrite methods for clausal and non-clausal theorem proving. In *Proceedings of the 10th International Colloquium on Automata, Languages, and Programming*, pages 331–346. EATCS, 1983.

[Her88] Maurice P. Herlihy. Impossibility and universality of wait-free synchronization. In *7th Symposium on Principles of Distributed Computing*. ACM, August 1988.

[Her90] Maurice P. Herlihy. A methodology for constructing highly concurrent data structures. In *ACM Symposium on Principles and Practice of Parallel Programming*, March 1990.

[HT90] Maurice P. Herlihy and Mark R. Tuttle. Wait-free computation in message-passing systems. In *9th Symposium on Principles of Distributed Computing*, pages 347–362. ACM, August 1990.

[Hue80] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, October 1980.

[Hul80] Jean-Marie Hullot. Canonical forms and unification. In W. Bibel and Robert A. Kowalski, editors, *Proceedings of the Fifth Conference on Automated Deduction*, pages 318–334. LNCS 87, Springer-Verlag, July 1980.

[HW90] Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, pages 463–492, July 1990. A preliminary version appeared in the proceedings of the 14th ACM Symposium on Principles of Programming Languages, 1987, under the title: *Axioms for concurrent objects.*

[JW90] M. D. Junkin and D. B. Wortman. Compiling concurrently. Technical Report CSRI-235, Computer Systems Research Institute, University of Toronto, Toronto, Ontario, CA, December 1990.

[KB70] Donald E. Knuth and Peter B. Bendix. *Simple Word Problems in Universal Algebras*, pages 263–297. Pergamon, Oxford, 1970.

[LAB+81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual.* LNCS 114, Springer-Verlag, Berlin, 1981.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[Lam80] Leslie Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1:77–101, October 1980.

[Lam83] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions of Porgramming Languages and Systems*, 5(2):190–222, April 1983.

[Lam89] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[Lam90] Leslie Lamport. A temporal logic of actions. Research Report 57, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, April 1990.

[Les] Pierre Lescanne. Completion procedures as transition rules + control. Centre de Recherche en Informatique de Nancy, France (unpublished).

[LF81] Nancy Lynch and Michael Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13:17–43, 1981.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* The MIT Press, Cambridge, MA, 1986.

[LH88] James R. Larus and Paul L. Hilfinger. Restructuring lisp programs for concurrent execution. In *Parallel Programming: Experience with Applications, Languages, and Systems*, pages 100–110, 1988.

[LT87] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, MIT Laboratory for Computer Science, Cambridge, MA, April 1987.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3), 1989.

[Luc87a] John Lucassen. Type and effects: Towards the integration of functional and imperative programming. Technical Report MIT/LCS/TR-408, MIT Laboratory for Computer Science, Cambridge, MA, August 1987.

[Luc87b] S. Lucco. Parallel programming in a vitual object space. In *Conference on Object Oriented Programming Systems, Languages, and Applications*, October 1987.

[Mar86] Ursula Martin. Doing algebra with REVE. Technical report, University of Manchester, Manchester, England, 1986.

[MNS87] Peter Moller-Nielsen and Jorgen Staunstrup. Problem-heap: A paradigm for multiprocessor algorithms. *Parallel Computing*, 4:63–74, 1987.

[Mus80] David R. Musser. On proving inductive properties of abstract data types. In *2nd ACM Symposium on Principles of Programming Languages*, pages 154–163, January 1980.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, August 1976.

[Pap79] C. H. Papadimitriou. The seriablizability of concurrent database updates. *Journal of the ACM*, pages 631–653, October 1979.

[PGH+89] C. D. Polychronopoulos, M. Gikar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchro-

nizing, and scheduling programs on multiprocessors. *International Journal of High Speed Computing*, 1(1):45–72, May 1989.

[Pon88] Carl Ponder. Evaluation of performance enhancements in algebraic manipulation systems. Technical Report UCB/CSD-88/438, Computer Science Division, University of California, Berkeley, CA 94720, 1988.

[PS81] G. E. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, April 1981.

[Rin89] G. A. Ringwood. Parlog85 and the dining logicians. *Communications of the ACM*, 31(1):10–25, January 1989.

[RR87] R. Ramesh and I.V. Ramakrishnan. Optimal speedups for parallel pattern matching in trees. In Pierre Lescanne, editor, *Proceedings of the 2nd International Conference on Rewriting Techniques and Applications, Bordeaux, France*, pages 274–285. Springer-Verlag, LNCS 256, May 1987.

[RV89] Eric Roberts and Mark Vandevoorde. Work crews: An abstraction for controlling parallelism. Technical Report 42, Digital Equipment Corporation Systems Research Center, Palo Alto, California, 1989.

[Sch88] James G. Schmolze. Parallel algorithms for knowledge representation. unpublished manuscript, 1988.

[SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[TSJ87] Chuck Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite Jr. Firefly: A multiprocessor workstation. Research Report 23, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, December 1987.

[VR90] Rakesh M. Verma and I.V. Ramakrishnan. Tight complexity bounds for term matching problems. *Information and Computation*, 1990.

[Win83] Jeannette M. Wing. A two-tiered approach to specifying programs. Technical Report MIT/LCS/TR-299, MIT Laboratory for Computer Science, Cambridge, MA, 1983.

[WW90] William E. Weihl and Paul Wang. Multi-version memory: Software cache management for concurrent B-trees. In *Proceedings of the Symposium on Parallel and Distributed Processing*, December 1990.