Structured Exception Semantics for Parallel Loops

A Thesis
Presented to
the faculty of the School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment
of the requirements of the Degree
Master of Science

by
Joel A. Winstead

January 2002

APPROVAL PAGE

This thesis is submitted in partial fulfillment of the requirements for the degree of
Master of Science

Joel A. Winstead

This thesis has been read and approved by the examining Committee:

Thesis Advisor

Accepted for the School of Engineering and Applied Science:

Dean, School of Engineering and
Applied Science

January 2002

Abstract

Concurrent languages have offered parallel loop constructs for some time to allow a parallel computation to be expressed in a simple and straightforward fashion. Modern programming languages include exceptions to allow for clean handling of errors or unexpected conditions, but few concurrent languages incorporate exception handling into their models for parallel loops. As a result, programmers that use parallel loops cannot use exceptions to simplify their programs. We present a semantics for handling exceptions in parallel loops that is predictable and that reduces to the familiar semantics for sequential loops. This semantics provides guarantees about the behavior of parallel loops even in the presence of exceptions, and facilitates the implementation of parallel algorithms. A Java library implementation of this semantics is presented, along with a description of a source-to-source translation.

To my parents

Contents

Chapter 1

Introduction

Exceptions generated in parallel loops create problems that can be difficult to resolve. Parallel loop semantics allow more than one iteration of a loop to be executed at a time. Because any statement can generate an exception, this makes it possible for more than one unhandled exception to be raised concurrently in the same loop, a situation that does not occur in sequential loops. However, the exception semantics of most languages do not allow more than one exception to be raised at a time. It is not clear how to deal with this situation in a consistent way. It may not be consistent with the language's exception model to propagate both exceptions; if only one exception is allowed, one must be chosen and the others ignored, and there may be consequences for ignoring an exception generated by the program. Although sequential loops simply stop executing once an exception occurs, there are several ways abnormal termination could be handled in the parallel case, and one must be chosen that is consistent and understandable.

Exception handling should be well-integrated with other parts of the language and exceptions should be handled in a consistent way regardless of the context in which they are generated. One proposed way of dealing with the problem of exceptions in parallel loops is simply to forbid them, and treat any exception that reaches the top level of a parallel loop as a fatal error. This is an unsatisfactory solution because exceptions in the context of a parallel loop are not handled in a manner that is consistent with the way exceptions in other contexts are handled, and it does not allow them to be caught by handlers higher in the call chain than the parallel loop, even if matching handlers exist. Ideally, an uncaught exception generated in a parallel loop should propagate out of the loop like any other exception, and be handled in a way that is consistent with the way exceptions are handled in other contexts.

Because the execution of statements in a parallel loop is not totally ordered, the order in which exceptions are generated in a parallel loop is not deterministic. For some machine models, it is difficult or impossible even to discover the order in which the exceptions occurred after the fact, so it is not clear how or if this nondeterminism can be resolved. Nondeterminism makes it difficult to assert strong postconditions of loops and functions, and this can make programs difficult to understand and debug, so eliminating nondeterminism or mitigating its effects where appropriate and possible is desired. Even in the event of exceptions, a loop may produce partial results that are still useful, so it is important to be able to make strong assertions about the state of the program after a parallel loop has terminated exceptionally. A good semantics for exception handling in parallel loops should provide a way to assert strong postconditions for both normal and exceptional loop terminations.

Nondeterminism occurs when the effects of a loop depend on the accident of how the loop happened to be scheduled and executed. A more useful semantics for exceptions in parallel loops can be developed if the effect of the loop depends on the structure of the loop, and not on the accident of its execution.

We propose a semantics for exceptions in parallel loops that does this by always propagating the exception that occurs structurally first in the loop, not the exception that occurs first in time. This removes the nondeterminism that results when more than one iteration of the loop throws an exception, and allows exceptions to be handled in a manner that is consistent with the way exceptions are handled in sequential loops. It allows stronger postconditions to be asserted about the result of the loop even in the case when an exception is thrown, because it allows the exception handler to know that all structurally prior iterations of the loop have completed without exceptions.

The next chapter describes concurrency constructs in high-level programming languages and discusses how concurrency can be integrated consistently into a structured language. Chapter 3 considers concurrency in Java, and ways in which concurrency could be better integrated into the language. Chapter 4 introduces the problem of exceptions in a concurrent language, and proposes a semantics for exceptions in parallel loops. In chapter 5, we provide a few examples of use of the new language constructs. In chapter 6, we explain how these ideas can be implemented in Java using a thread library and a source-to-source translator.

Chapter 2

Concurrency Constructs

Programming languages have contained constructs that express concurrency for some time. Concurrency constructs are useful not only for writing parallel algorithms that allow programs to be executed more efficiently, but also for expressing inherently concurrent algorithms in a clear way. Having good mechanisms for expressing concurrency is important for a modern programming language, and many solutions have been proposed.

2.1 Ada Tasks

Ada tasks present a simple construct for providing concurrency in a structured way. An Ada task is declared in a similar way to a package, may contain its own declarations, and contains a block of code that is to be executed concurrently with the outer (or master) block that declared the task. Tasks may also declare entries, which may be used for synchronization. Outside code may call an entry in the same way that it might call a procedure, and will block until the task has processed the entry. The task body is responsible for accepting and handling entry calls [1].

Ada tasks are useful for describing conceptually separate tasks that are to be executed in the same program, and are particularly useful for implementing algorithms which are inherently concurrent and require communication and synchronization with other tasks.

2.2 Fork and Join

A simple and popular construct for expressing concurrency in a language is to have a `fork` operation that starts a new process which runs concurrently with the current process. The new process is written as a separate procedure from the one that executes the `fork` statement; after the `fork` operation, the original process continues with the next statement, and the new process executes the specified procedure concurrently with the original process. The `fork` operation returns an identifier of the newly-created process to the parent. When the forked procedure terminates, there is no implicit notification of the original process; however, processes can synchronize using the `join` operation on the ID of another process, which causes the calling process to block until the specified process is complete.

This construct for expressing concurrency is often implemented in a library rather than as a syntactic construct of the language itself, as is done in the POSIX Threads interface [2], the Win32 API [3], and Java [4]. Other languages implement the `fork`/`join` construct as language keywords, such as the `spawn` keyword in Compositional C++ [5].

2.3 Parallel Blocks

Dijkstra's `parbegin`/`parend` construct [6] is a simple syntactic construct to express concurrency in block-structured languages that has served as a model for concurrency constructs in many languages. In its simplest form,

```
parbegin
    $S_1$
    $S_2$
    ⋮
    $S_N$
parend
```

the N statements begin and execute concurrently as separate threads of execution when the program reaches the `parbegin` statement. All N statements must complete before the main program resumes execution with the `parend` statement. The order in which the N statements appear in the block is irrelevant. Statements that alter the flow of control in the program, such as return statements or jumps out of the block or to other statements in the block, are not allowed, in part because it is not clear what they would mean.

The actual manner in which the statements execute, whether in parallel, by interleaving, or in some sequential order, is not specified. The construct does guarantee, however, that no statement within the construct will starve indefinitely: progress will be made as long as there is at least one statement in the block that is not blocked and can make progress. This allows the construct to be implemented in a variety of ways, such as separate tasks on different processors of a parallel machine, or using lightweight threads on single-processor machines, or by using a compiler that analyses the loop and determines a sequential order that is safe and deadlock-free.

Because the `parbegin`/`parend` construct is part of a block-structured language (it was originally proposed in the context of Algol 60), parallel blocks may be nested. This effectively creates a tree-structured hierarchy of threads, in which interior nodes block waiting for their children to complete, and only leaf nodes perform computation. This structure allows flexibility and facilitates computation on irregular data.

This construct provides a clean way to express which parts of the program can execute concurrently and which parts must be executed sequentially, and because it is a part of the language, it is easier to read and analyze than calls to an external thread library. Because the statement following the construct cannot begin until all statements within the construct have completed, it is possible to assert a meaningful postcondition of the block.

Forms of the `parbegin`/`parend` construct have appeared in Algol68 [7], pSather [8], Compositional C++ [5], and many other languages.

2.4 Parallel Loops

A parallel loop is a `for` loop in which the iterations of the loop execute concurrently rather than sequentially. Like the `parbegin`/`parend` construct, when the program reaches a parallel loop it spawns multiple threads to execute the iterations of the loop, and requires that all threads complete before the program continues with the next statement. Each thread within the loop receives its own copy of the iteration variable so that the threads can perform independent computations. For example, the following loop computes the vector sum of two arrays in parallel:

```
parfor (int i=0; i<N; i++)
     a[i] = b[i] + c[i];
```

Each iteration of the loop has its own value for *i* and operates on a different part of the data. Assuming that *a* does not share storage with *b* or *c*, all iterations of the loop can execute simultaneously.

The program does not return to the surrounding block and continue with the next statement until all iterations have completed. This particular loop can be executed efficiently in parallel because it has no data dependencies between iterations.

Parallel loops may be either synchronous or asynchronous. Synchronous loops imply synchronization between statements in different iterations of the loop; the form of the implied synchronization varies from language to language, but all generally require each iteration to execute the same statement at the same time. A Fortran `forall` loop containing a single assignment statement as its body, for example, requires that the value of the right-hand sides of the assignments must be computed first for every iteration of the loop before any actual assignments are made, thus avoiding potentially harmful effects. This semantics is particularly well-suited to vector machines.

Asynchronous loops do not have any implicit synchronization between statements in different iterations. This allows the iterations of the loop body to proceed independently of one another. Explicit synchronization statements can be used to provide a stronger ordering on loops that require it. Asynchronous loop semantics are well-suited to systems that implement parallelism as threads that are scheduled independently. Asynchronous loops have appeared in Compositional C++ [5], Modula-3* [9], and other parallel languages.

The asynchronous loop semantics used by Compositional C++ can be defined more formally in terms of `fork` and `join` statements. The parallel loop resembles a C-style `for` loop, defined by an index variable, condition expression, update statement, and loop body. For the time being, none of the parts of the loop is permitted to generate exceptions, and the use of an explicit index variable is required.

```
parfor ( index_var = initial_value ;
              condition_expression ;
              update_statement ) {
     loop_body;
}
```

The meaning of this loop is expressed in terms of the familiar `for` loop construct, and the `fork` and `join` constructs:

```
i = 0;
for ( index_var = initial_value ;
              condition_expression ;
              update_statement ) {
     ids[i++] = fork body( index_var );
}
for (j = 0; j < i; j++) {
     join ids[j];
}

body( index_var ) {
     loop_body;
}
```

This first loop executes the control portion of the parallel loop, determines which index values to use, and forks threads to execute the body of the loop for each value of the index variable. After all iterations have been started, the second loop waits for them to complete by joining each iteration in turn. The body of the loop is separate, and each iteration of it has its own local copy of the index variable to work with.

This semantics places a partial order on the execution of the loop. The body for a particular iteration cannot execute until the condition for that iteration has been evaluated. The condition C_{i+1} for iteration $i + 1$ cannot be executed until the update statement U_i for iteration i has completed, which in turn depends on the condition expression C_i for iteration i. No order is imposed between statements in the body from different iterations. Each iteration i must execute its last statement $B_{i,N}$ before the loop terminates (T).

Some implementations, such as the `forall` loops in Fortran 95 [10], require that the number of iterations of the loop, and their index values, be known when the loop starts; this simplifies the implementation and allows all iterations to start immediately, at the cost of flexibility. Other parallel loop constructs, such as the `parfor` construct in Compositional C++, do not have this restriction, and have the control portion of the loop execute sequentially while spawning threads to execute the body of the loop [5]; this allows loops where the number of iterations is not known in advance, or that do not have integer indices.

Because many numerical algorithms perform the same operations repeatedly over arrays of data, `parfor` loops are a natural way to express that these operations can be performed in parallel to improve efficiency.

2.4.1 Extending Java with Parallel Language Constructs

Concurrency in Java is expressed using objects. Each thread of control in the program is implemented as a separate object that either extends the `Thread` class or implements the `Runnable` interface and has a controlling `Thread` object. Every object in Java has its own associated monitor which is used for synchronization. There are no explicit parallel constructs equivalent to parallel

$$\begin{tikzcd}
I \arrow[d] & & & & & \\
C_0 \arrow[r] \arrow[d] & B_{0,0} \arrow[r] & B_{0,1} \arrow[r] & B_{0,2} \arrow[r, dotted] & B_{0,M} \arrow[ddddr] & \\
U_0 \arrow[d] & & & & & \\
C_1 \arrow[r] \arrow[d] & B_{1,0} \arrow[r] & B_{1,1} \arrow[r] & B_{1,2} \arrow[r, dotted] & B_{1,M} \arrow[ddr] & \\
U_1 \arrow[d] & & & & & \\
C_2 \arrow[r] \arrow[d, dotted] & B_{2,0} \arrow[r] & B_{2,1} \arrow[r] & B_{2,2} \arrow[r, dotted] & B_{2,M} \arrow[r] & T \\
C_N \arrow[r] \arrow[d] & B_{N,0} \arrow[r] & B_{N,1} \arrow[r] & B_{N,2} \arrow[r, dotted] & B_{N,M} \arrow[ur] & \\
U_N \arrow[d] & & & & & \\
C_{N+1} \arrow[uuurrrrr] & & & & &
\end{tikzcd}$$

Figure 2.1: Partial order of `parfor` loop execution

loops or `parbegin`/`parend`, and each thread within an application runs at the same level: there is no parent-child hierarchy of threads. This model is well suited for writing programs with separate, unrelated tasks that execute concurrently, such as having one thread compute in the background while another thread handles the user interface. This technique is used mainly to separate the tasks in an application and to reduce latency in user interfaces. This thread model is not well suited for data parallelism, where many threads are to perform the same or similar operations on different pieces of data as part of the same task, because the constructs for declaring, starting, and joining tasks are not control flow constructs that can easily be used in an algorithm description.

A `for` loop in Java consists of an initialization statement, a condition expression, an update statement, and a loop body. The initialization statement is executed before any statement begins; it may contain a variable declaration that has the scope of the entire loop. During each iteration, the condition expression is evaluated first. If it is true, the body and update statements are executed, in sequence. This process is repeated until either the condition evaluates to false, in which case the loop terminates normally, or one of the parts of the loop terminates abnormally, in which case the loop terminates abnormally for the same reason and does not execute any further statements. There is an implicit requirement that all iterations of the loop must terminate before the loop as a whole terminates; this requirement follows from the fact that the iterations run sequentially [11].

This semantics can be extended to allow asynchronous parallel execution of the loop by eliminating the requirement that the body complete before the update statement and the next iteration begins. This allows the control portion of the loop to execute sequentially, while the iterations of the body execute concurrently. Because more than one iteration of the body executes at a time, there must be a way to distinguish between iterations, so we also add a requirement for an explicit iteration variable, declared in the initialization statement. Each iteration of the body of the loop gets a private copy of the iteration variable when it is started, and the body can use this value to determine what part of the task to perform. The copying allows the control portion of the loop to continue without interfering with the body's use of the iteration variable, and allows multiple iterations of the loop body to execute concurrently while providing a way for them to be distinguished so that they can perform their respective tasks.

In order to allow a postcondition to be asserted about the resulting state of the program after loop termination, we still require all iterations of the loop body to terminate before the loop as a whole terminates, even if the loop terminates abnormally. This is necessarily the case for sequential loops because only one statement executes at a time, but it must be stated as an explicit requirement for parallel loops, because otherwise it may be possible for the statements following the loop to start while some statements within the loop continue to execute. This situation would make the state of the program unknown at the time the loop terminates, and could lead to unpredictable behavior. To avoid this, we require all iterations of the loop to terminate before the program continues with the first statement following the loop. This allows a postcondition to be asserted about the state of the program after the loop has terminated, which would not be possible if some of the loop's iterations were still executing.

Nondeterminacy can also result if there are data dependencies between different iterations of the loop body, or between the body and the control portion of the loop. Although in some cases, a compiler could detect these problems and insert synchronization statements to eliminate them, the approach that Java takes for this sort of problem in general is to place the burden on the programmer to use appropriate concurrency control constructs, such as the `synchronized` statement, to prevent this sort of problem. The same care should be taken when using the proposed parallel loop extension.

Because exceptions may be raised within loops, it is important to have a mechanism to handle them. This could be done by enclosing the loop within Java's `try`..`catch`..`finally` construct, but this would lose information about which iteration it was that raised the exception, because the iteration variable would no longer be in scope. In order to preserve this information, we propose allowing `catch` and `finally` clauses to be attached directly to the loop construct, and allow the iteration variable declared in the initialization portion of the loop to remain in scope for the exception handlers contained there. If an exception occurs in the evaluation of the condition, update statement, or loop body, and there is a matching `catch` clause attached to the loop, control is passed to the handler with the current value of the iteration variable intact.[1] This allows an exception handler to use the value of the iteration variable when it handles the exception. If no matching `catch` clause is found attached to the loop, the exception propagates using the normal rules for exception handling, and the value of the iteration variable is lost. This construct could be used for either sequential loops or parallel loops, and is independent of any particular exception semantics. The precise semantics for exceptions in a Java parallel loop are described in the next chapter.

[1]If the initialization statement raises an exception, any attached `catch` clauses or `finally` clauses must be ignored, and the exception will propagate normally. This is because the iteration variable will not be defined, and if the `catch` or `finally` clauses attached to the loop attempt to use this undefined variable, a code safety problem would result.

Chapter 3

Exception Semantics for Concurrency Constructs

Although parallel loops have been implemented in a number of languages, few have been implemented in languages that have exceptions, and even fewer have attempted to address the semantics of an uncaught exception that occurs within a parallel loop. Exceptions in parallel loops introduce several difficult situations that must be addressed, especially when exceptions occur within more than one iteration of the loop. When an exception occurs within a sequential loop, the loop simply stops executing and the exception is propagated to the calling block; because of sequential loop semantics, the handler may assume that this was the only exception that occurred, and that all iterations up to the exception completed normally. When an exception occurs within a parallel loop, the iteration that generated the exception necessarily stops executing, but it is less clear what to do about the other iterations, and how or if the exception should be propagated. The question of what to do if more than one iteration of the loop throws an exception is a difficult issue as well, because the exception semantics of most languages do not allow two exceptions to be raised simultaneously.

3.1 Goals for Exception Semantics

A simple approach to the problem of exceptions in parallel loops is to ignore uncaught exceptions in concurrent code, or to forbid them altogether. The designers of Ada, when confronted by the issues exceptions raise in a concurrent context, chose to ignore uncaught exceptions that reach the top level of a task. If an uncaught exception occurs in an Ada task, the task terminates without handling the exception or passing it on to an outer block that can handle it; this policy was chosen to prevent the problems that would result if an exception were passed to a parent task asynchronously [12]. The designers of pSather chose to forbid exceptions in a concurrent context [13] [14]. These solutions deny concurrent programmers the expressive and robustness benefits of exceptions.

Programmers should be able to use exceptions in concurrent constructs as they would normally in the programming language, and be able to reason about their behavior. A good semantics for uncaught exceptions in parallel loops should be consistent. If the exception which is propagated from the loop is chosen nondeterministically, this could make the program difficult to reason about. Some forms of nondeterminism are unavoidable and even desirable in an asynchronous loop, because the parallel loop semantics impose only a partial order on the statements in the loop; however, the ability to predict and reason about the exception, if any, produced by the loop would make programs easier to write.

In order to recover from an exception in a loop, it is important to know the state of the program after loop termination. A good exception semantics should allow a strong postcondition to be asserted about the state of the program after an exception has been thrown. This postcondition can then be assumed by the exception handler, which can use the information in its recovery.

An exception semantics should also enable a clear termination condition for the loop. If an iteration within a loop terminates with an exception, and other threads within the loop depend on the normal completion of that iteration, a deadlock could result. A good exception semantics should provide a way to prevent this sort of problem from occurring whenever possible.

The original `parbegin`/`parend` block was proposed in the context of Algol 60, which did not have exceptions. However, the proposal did forbid the use of return statements and jumps out of the block. Some programming languages deal with the exception issue the same way by forbidding exceptions in parallel blocks and loops. pSather is a parallel language which takes this approach. If an exception occurs within the body of a parallel loop or block and is not handled inside the block, it is treated as a fatal error and the program halts immediately. As a consequence, all exception handling for parallel constructs must occur within the parallel construct, and an exception in a parallel construct cannot automatically be passed up the call chain to a higher level that has a handler for it [8].

3.2 Exception Propagation

Java, C++, and other object-oriented languages which have exceptions can only raise one exception at a time. In a parallel loop, however, it may be possible to raise multiple exceptions within the same block of code. It is important to have a clear semantics for what should happen in this case.

One possibility is to handle each exception in the loop as it happens, while allowing other iterations in the loop to continue, possibly generating more exceptions. This solution would create consistency problems because it would allow code within the loop to run at the same time as code in outer blocks, and could even result in destroying stack frames that are part of the loop's context. This would create code safety problems, and is inconsistent with the loop semantics defined in the previous chapter, because the semantics state that all iterations of the loop must complete before any outside exception handler begins.

Another possibility is to merge all of the exceptions together into a single exception that contains an array storing the exceptions generated by each iteration of the loop. This allows all exception information to be kept. The cost of this approach is that each iteration of the loop must be allowed to run to completion in order to determine which exceptions will be thrown. This could result in deadlock if there are dependencies between events in different iterations of the loop. Code written to catch and handle exceptions would need to be more complicated, because either `catch` expressions would need to be able to match patterns in the array, or some mechanism would be required to apply the chain of exception handlers to each exception in the array. This would be particularly complex if some exceptions were handled higher in the call chain, while others were handled at lower levels.

A scheme for collapsing multiple exceptions into a single exception in a Modula-3* parallel loop is described by Heinz [15]. If a single exception is generated by the loop, or if all exceptions in the loop are equal, a single copy of the exception is propagated out of the loop. If the exceptions are not all equal, a special exception is generated to note the inconsistency. This approach allows multiple exceptions cases to be handled in languages which can only raise a single exception at a time. However, it requires all iterations of the loop to complete, and surrounding code must be able to handle the case inconsistent exceptions.

In many cases, if multiple exceptions occur in a parallel loop, the exceptions are all related to the same problem, and it is only necessary to propagate one of the exceptions to ensure a proper recovery; other exceptions can be discarded. Philippsen and Blount describe an implementation of asynchronous parallel loops in Java that uses this approach [16] [17]. The first exception that occurs in time is propagated out of the loop; other iterations are canceled, and any exceptions they generate are ignored. This approach loses some information and introduces nondeterminism in the exception that is returned, but keeps within the single-exception model of the language, does not require every iteration of the loop to complete.

Our approach is to return not the first exception that occurs in time, but the exception that occurs structurally first in the loop. This removes the nondeterminism in the exception that is propagated, and loops can be written in a way that guarantees that the most important exception, if any, will be the one that is propagated out of the loop. Our semantics requires structurally prior iterations of the loop to complete so that any structurally earlier exceptions can be obtained. Structurally later iterations, however, can be canceled, thus eliminating deadlock in situations where no iteration of the loop waits for an event in a structurally later iteration.

3.3 Cancellation and Loop Termination

In order to determine what exception semantics is most appropriate for parallel loops, the termination condition imposed by each exception semantics must be considered. A straightforward solution is to require all iterations of the loop to terminate, even in the event of an exception. This approach is used by Heinz in Modula-3* loops [15]. In cases where some iterations of the loop wait or depend on events that occur in other iterations, the loop may deadlock. In particular, if an iteration of the loop terminates with an exception before creating an event that some other iteration is waiting for, the loop will deadlock because the second iteration is waiting for an event that will never occur. This situation can be created by the use of mutual exclusion for a shared resource, which is common in asynchronous parallel programs. Requiring all iterations of the loop to terminate even in the event of an exception is practical if it is known that there is no communication or explicit synchronization between iterations, but many loops do not satisfy this condition, so this solution cannot be used for a general parallel loop.

Another solution is to cancel some iterations of the loop in the event of an exception in order to guarantee that the loop terminates[1]. One approach is to cancel all iterations in the loop other than

[1]Thread cancellation itself can be dangerous; the implementation chapter describes how this can be done safely in Java

the one that generated the exception. This approach guarantees that the loop will terminate in the event of an exception. Because it cancels some iterations, this cancellation strategy is not consistent with any approach to exception propagation that requires all iterations to terminate. In addition, because this approach cancels both structurally prior and structurally later iterations, it cannot be used if the semantics requires the structurally first exception from the loop to be propagated. This approach is consistent with an exception propagation semantics that requires the first exception in time to be propagated.

Because canceling structurally prior iterations is inconsistent with a semantics that propagates the structurally first exception, we adopt a cancellation strategy that cancels structurally later iterations only. Because it requires some iterations of the loop to complete, it can result in deadlock if an iteration waits for an event that is generated by a structurally later iteration; however, if no iteration waits for an event generated by a later iteration, this cancellation strategy guarantees that the loop will terminate in the event of an exception, provided that the structurally prior iterations themselves terminate. This approach is consistent with a semantics that propagates the structurally first exception out of the loop.

Consider the following loop, which computes Pascal's triangle in parallel:

```
int triangle[N][N];
Flag flags[N][N];
parfor (int i=0; i<N; i++) {
     triangle[i][0]=1;
     flags[i][0].set();
     parfor (int j=1; j<i; j++) {
          flags[i-1][j].check();
          flags[i-1][j-1].check();
          triangle[i][j] = triangle[i-1][j]+triangle[i-1][j-1];
          flags[i][j].set();
     }
     if (i>0) {
          triangle[i][i]=1;
          flags[i][i].set();
     }
}
```

Each iteration of the inner loop depends on two structurally previous computations and will block until they are complete. This loop makes use of monotonic flags to make sure that each computation waits until the data that it depends on is available. If a flag is not yet set, the `check` operation on the flag causes the caller to block until some other thread performs the `set` operation on the flag. If the flag has already been set, the `check` operation returns immediately.

Suppose iteration $i = 5, j = 5$ within the loop terminates with an exception before setting its flag. If no cancellation of other iterations occurred, the loop would not complete because iteration $i = 6, j = 5$ and iteration $i = 6, j = 6$ each depend on the setting of `flag[5][5]` and would block indefinitely. If all iterations of the loop were interrupted, the iterations waiting for `flag[5][5]` would receive the interruption and terminate, allowing the loop as a whole to terminate, although there is no guarantee that any partial results would be available. If all structurally later iterations of the loop (i.e. all iterations with $i > 5$ or $j > 5$) were interrupted, the iterations that depend on `flag[5][5]` either directly or indirectly will terminate, while allowing structurally earlier iterations, which do not depend on `flag[5][5]`, to complete.

Note that if the outer loop counted down from $i = N$ to $i = 0$, the condition that no iteration depends on a structurally later iteration would not hold. Iteration $i = 6, j = 6$ would be structurally prior to iteration $i = 5, j = 5$, yet would still depend on `flags[5][5]`. If this were the case, the loop would not terminate if only structurally later iterations were interrupted. The loop would still terminate if all iterations were interrupted, however. The policy of canceling structurally later iterations is useful only if the loop is written so that the condition holds and the loop can terminate when structurally later iterations are interrupted.

3.4 Loop Postconditions

In order to write correct programs, it is important be able to make assertions about the state of the program after loop termination, even in the event of an exception. The exception semantics directly affects the kinds of postconditions that can be asserted about a loop that terminates with an exception.

Because different iterations of a parallel loop may share some context and data, and there is no explicitly defined order between events in one iteration of the loop and events in another, the postcondition that can be asserted of one iteration of the loop is not necessarily the same as the strongest postcondition that could be asserted of the same sequence of statements if executed in a sequential context. The postcondition for each iteration of the loop must be true for any possible interleaving of the different iterations of the loop and any potential interactions between them, and may not assume anything about the progress of other iterations. For example, consider the following loop:

```
    parfor (int i=0; i<100; i++)
        A[i] = i*i;
```

If this loop were executed sequentially, the postcondition of iteration $i = 0$ would be $A[0] = 0$ with all other elements of A unchanged, or $A[0] = 0 \wedge \forall i : i > 0 : A[i] = A_{pre}[i]$. It it were executed concurrently, however, the strongest postcondition that can be asserted for iteration $i = 0$ is $A[0] = 0 \wedge \forall i : 0 < i < 100 : (A[i] = A_{pre}[i] \vee A[i] = i^2) \wedge \forall i : i >= 100 : A[i] = A_{pre}[i]$. This weaker postcondition takes into account the fact that the other iterations may not have completed by the time iteration $i = 0$ has. If different iterations of the loop attempted to write different values to the same variable without explicit synchronization, the postconditions for all iterations of the loop must account for each possible ordering of the writes.

The loop guarantee is the strongest condition that can be asserted of the loop at all times, and must allow for every possible interleaving of iterations, and every possible partial result. For this loop, this is $(\forall i : 0 \leq i < 100 : A[i] = A_{pre}[i] \vee A[i] = i^2) \wedge (\forall i : i \geq 100 : A[i] = A_{pre}[i])$, expressing the fact that any combination of the iterations may have completed.

The postcondition of the loop as a whole is the conjunction of the loop guarantee and the postconditions of all iterations of the loop that are known to have completed. For the above example, this is $(\forall i : 0 \leq i < 100 : A[i] = i^2) \wedge (\forall i : i \geq 100 : A[i] = A_{pre}[i])$, assuming that all iterations complete. The terms in the individual iteration postconditions that expressed uncertainty about the progress of the other iterations are absorbed when the fact that each iteration has completed is included in the postcondition.

If one or more iterations of the loop terminate with exceptions, the postconditions for these iterations may not be true, and they cannot be used in the loop postcondition. The cancellation strategy may prevent further iterations from completing, and the postconditions of these iterations may not be used either. In addition, although the exception propagation policy does not affect which iterations will complete, it may limit the knowledge the exception handler has of the completion status of the loop. For any iteration that does not complete, the strongest postcondition that can be asserted is the uncertainty about how much progress the iteration made, which includes each potential partial result of the iteration; this condition of uncertainty is included in the loop guarantee.

If the first exception thrown in time is propagated from the loop, and other exceptions are ignored, an exception handler may not assume anything about the termination of other iterations of the loop, regardless of the termination strategy used. Even if the cancellation strategy allows other iterations of the loop to complete, it is possible that some of them may terminate exceptionally, and it would not be safe for the exception handler to assume anything about them. In this case, no iteration of the loop can be assumed to have completed, and no postcondition can be asserted of the loop that is stronger than the loop guarantee.

If the structurally first exception is propagated from the loop, and other exceptions are ignored, then all iterations structurally prior to the one that terminated abnormally are known to have completed. However, nothing can be asserted about the completion status of any other iterations, because any of them may have terminated exceptionally. If this exception propagation policy is used, the strongest postcondition that can be asserted of the loop as the exception handler is entered is the conjunction of the loop guarantee with the postconditions of all iterations structurally prior to the one that terminated with the exception. In the above example, if an exception occurred in iteration $i = 50$, the exception handler could assume $(\forall i : 0 \leq i < 50 : A[i] = i^2) \wedge (\forall i : 50 \leq i < 100 : A[i] = A_{pre}[i] \vee A[i] = i^2)$.

3.5 Structured Exception Semantics for Parallel Loops

Because propagating the structurally first exception in the loop yields the most useful loop postcondition, and because canceling structurally later threads in the loop increases the class of problems for which the loop terminates in the exception case, we propose a semantics for parallel loops that propagates the structurally first exception. Under this semantics, the effect and postcondition of a loop that terminates with an exception is determined by the structure of the loop, not by the accident of its execution.

In order to define a `parfor` loop in Java that uses this semantics, we must modify the Java language semantics for a `for` loop. The Java Language Specification [11] gives the following description of the execution of a sequential `for` loop:

- If the *Expression* is present, it is evaluated, and if evaluation of the *Expression* completes abruptly, the `for` statement completes abruptly for the same reason. Otherwise, there is then a choice based on the presence or absence of the *Expression* and the resulting value if the *Expression* is present:
  - If the *Expression* is not present, or it is present and the value resulting from its evaluation

is `true`, then the contained *Statement* is executed. Then there is a choice:

  * If execution of the *Statement* completes normally, then the following two steps are performed in sequence:
    - First, if the *ForUpdate* part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If evaluation of any expression completes abruptly for some reason, the `for` statement completes abruptly for the same reason; any *ForUpdate* statement expressions to the right of the one that completed abruptly are not evaluated. If the *ForUpdate* part is not present, no action is taken.
    - Second, another `for` iteration step is performed.
  * If execution of the *Statement* completes abruptly, see §14.13.3 below.

- If the *Expression* is present and the value resulting from its evaluation is `false`, no further action is taken and the `for` statement completes normally.

In order to achieve parallel execution of the loop, we must modify this definition so that instead of executing the *Statement* directly, it starts a thread which executes the *Statement* concurrently. The *Statement* thread receives its own copy of the iteration variable so that it may execute independently of the loop control and the other iterations. In addition, we must add a rule that states that when the loop completes, either normally or abruptly, it must wait for all concurrent *Statement* threads to complete.

These changes allow the loop to execute concurrently, but further changes are needed to handle abrupt termination in the event of an exception. Section 14.13.3 of the specification describes abrupt termination of a `for` loop in the case of a `break` or `continue` statement, and states that "If execution of the *Statement* completes abruptly for any other reason, the `for` statement completes abruptly for the same reason." For simplicity, we do not allow `break`, `continue`, or `return` statements in `parfor` loops, although they could be introduced in ways consistent with our semantics. Hence, the only form of abrupt termination we consider here is through exceptions. In the event of an abrupt termination, the control portion of the loop should complete immediately, and no further *Statement* threads should be started. Any running *Statement* threads that were started after the *Statement* that generated the exception should be interrupted, which will allow them to terminate cleanly. Once all *Statement* threads have completed, the loop should terminate by propagating the structurally earliest exception.

Together, these changes produce the following specification for a Java `parfor` loop:

- First, the `parfor` control loop is executed:

  If the *Expression* is present, it is evaluated, and if evaluation of the *Expression* completes abruptly, the `parfor` control completes abruptly for the same reason. Otherwise, there is then a choice based on the presence or absence of the *Expression* and the resulting value if the *Expression* is present:

  - If *Expression* is not present, or it is present and the value resulting from its evaluation is `true`, and no existing *Statement* thread has terminated exceptionally, then a thread is started to execute the contained *Statement* concurrently with the `parfor` control. A copy is made of the iteration variable, the new thread uses the copy.
    - Then the `parfor` control performs the following two steps in sequence:
      - First, if the *ForUpdate* part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If evaluation of any expression completes abruptly for some reason, the `parfor` control completes abruptly for the same reason; any *ForUpdate* statement expressions to the right of the one that completed abruptly are not evaluated.
      - Second, another `parfor` control step is performed.
  - If the *Expression* is present and the resulting value of its execution is `false`, no further action is taken and the `parfor` control completes normally.

- If a *Statement* thread terminates abruptly, all running *Statement* threads that were started after the one that terminated abruptly are interrupted. The `parfor` control then completes without starting any new *Statement* threads.

- When the `parfor` control completes, it must wait for all of its *Statement* threads to complete before terminating. If any errors or exceptions occurred in the execution of either the `parfor` control or any *Statement* threads, the error or exception from the earliest iteration is propagated as the reason for the `parfor` statement's termination.

To illustrate the semantics, consider the array assignments example introduced in §3.4. If A is an array of size 50 instead of size 100, any attempt to access elements 50 through 99 will result in an `ArrayBoundsException`. Because the iterations of the loop execute in parallel, however, it is not possible to know in advance which iteration will cause an exception to be thrown first, nor is it possible, once that statement has been reached, to know which iterations of the loop have already completed or how much of the array has been filled.

With the proposed semantics for abnormal loop termination, although it cannot be known which iteration will generate the first exception, we do know that the structurally first exception will be the one that is propagated, and that all lower iterations will complete before the loop terminates. This means that even though we cannot know whether iteration $i = 50$ or iteration $i = 60$ will generate an exception first, we do know that the `ArrayBoundsException` generated by iteration $i = 50$ will be the one propagated by the loop, and the handler for that exception may safely assume that iterations $i = 0..49$ have completed and that the array up to the point the exception occurred has been filled, and this can be asserted as a postcondition of the loop.

The result of an exception with this semantics is consistent with the semantics of a normal sequential `for` loop. In a normal `for` loop, only one iteration of the loop executes at a time, and the iterations execute in order. If an exception is thrown in one iteration of the loop, the loop terminates immediately and propagates the exception to the calling block, and no further iterations of the loop are executed. The program may safely assume that all iterations up to the iteration that threw the exception have completed normally, and the handler for the exception can use this information when recovering. Even if there are additional problems that would have caused higher-numbered iterations of the loop to throw exceptions, only the first such problem is caught and reported. Our proposed exception semantics for parallel loops shares these properties: the lower-numbered iterations of the loop are guaranteed to have terminated normally, and the exception that is propagated is the one from the lowest-numbered iteration that threw an exception.

3.6 Handling Exceptions from Parallel Loops

In order for an exception handler to be able to make use of the loop postcondition described above, it must know how much of the loop completed and what iteration it was that generated the exception. To support this, we allow `catch` and `finally` clauses to be attached directly to `parfor` statements. The iteration variable for the loop remains in scope within these clauses. Because more than one copy of the iteration variable exists in the parallel loop semantics, we must specify what the iteration variable in the exception handlers refers to.

If an exception occurs within the body of the parallel loop, the value of the iteration variable in the handler should be the same as the value of the iteration variable in the instance of the body that raised the exception. If an exception is generated in the sequential control portion of the loop, then the handlers should see the current value of the iteration variable from the control portion when it stopped. This allows the handler to know which iteration caused the exception, so that it can use this information when recovering from the exception.

If an exception occurs in the initialization statement of the loop, which declares the iteration variable, the variable is undefined, and the `catch` and `finally` clauses of the loop cannot be allowed to execute because the result would be undefined and could introduce a code safety issue. For these purposes, the initialization statement can be considered to be outside of the loop, and outside of any attached `catch` and `finally` clauses. Exceptions generated by the initialization statement can still be handled, but they must be handled outside of the loop.

If no `catch` clause attached to the loop handles the exception, the exception will propagate up the call chain normally, and the information about which iteration caused the exception will be lost. An alternative might be to include the information about which loop and which iteration threw the exception in the exception object itself. However, this is undesirable. It would require a fundamental change to Java's exception model which would affect much more than just loops, and that would make the parallel exception semantics inconsistent with all normal Java programs. Furthermore, it is unlikely that the information about which iteration caused the exception would be useful outside of the context of the loop, because the exception may propagate far up the call chain to handlers that have no special knowledge of the loop or how to use the value of the iteration variable to recover from the loop. For these reasons, it is acceptable to discard the information about which iteration caused the exception if there is no handler directly attached to the loop that can use this information. Even when this information is discarded, it still is useful to propagate the exception up the call chain, where there may be a handler that can deal with the exception without knowledge of the loop or iteration variable.

Although an exception handler inside the loop body, rather than attached to the loop itself, would also have access to the value of iteration variable, such a handler would not have the knowledge that all structurally prior iterations of the loop have completed, and would not be able to use this information when attempting to recover from the exception. The possibility that an exception handler could use both the value of the iteration variable and the knowledge that structurally prior iterations have completed justifies allowing a special handler to be attached to the loop itself.

3.7 Safe Thread Cancellation

The semantics require that all iterations structurally prior to an iteration that causes an exception must be allowed to complete, because they may also throw exceptions, and we want to propagate the exception that is structurally first, not first in time. Any higher iterations that have not yet started will never be started, and higher iterations that are already running are interrupted. The purpose of this is to cancel those iterations that occur logically after the exception and are therefore invalid.

Cancellation of running threads is difficult to do safely in an object-oriented language [13]. Simply terminating a thread with no mechanism to allow cleanup is unsafe and deadlock-prone, because the thread may be holding locks or may have temporarily placed an object in an inconsistent state. For this reason, a mechanism to allow the thread to exit cleanly is needed.

One mechanism for terminating a thread provided by early versions of Java allowed an exception to be delivered to a thread asynchronously, though the `Thread.stop()` method. This is dangerous, however, because it can deliver an exception to code that was not designed to handle that particular exception. Although it is theoretically possible to write code that can handle asynchronous exceptions, it is very cumbersome to read and very difficult to get correct, particularly when the exception is delivered in a `finally` clause or `synchronized` block. In any case, the mechanism is not guaranteed to cause the thread to stop: the thread's code could be written in a way to trap such exceptions and ignore them. Because of these problems, Sun chose to deprecate this feature of Java [18].

Instead, Sun recommends that running threads be canceled using the `Thread.interrupt()` mechanism, which is safe because it causes the thread to be interrupted only at well-defined points in the code. Certain library routines can generate an `InterruptedException`, which is a checked exception which must be caught or declared. Other code can explicitly check to see if it has been interrupted by using the `Thread.interrupted()` call, and it is the responsibility of the programmer to make sure that the thread exits cleanly. Although it is possible for code to ignore the interruption and continue running, this is not worse than the `Thread.stop()` method, which cannot provide this guarantee either, and is much harder to use safely [4].

For these reasons, we attempt to terminate running iterations of the loop using the interrupt mechanism. Because poorly written code could ignore the interrupt mechanism, the semantics cannot provide a guarantee that canceled iterations will terminate promptly, or even that they will terminate at all; however, correct Java code should terminate when interrupted. Because the loop semantics require all iterations of the loop to terminate before the loop itself terminates, the semantics cannot guarantee for arbitrary loops that the loop as a whole terminates either, even when a loop iteration raises an exception. Care must be taken to make sure that loops are written such that they will terminate cleanly, even in the presence of exceptions. This is already the case for sequential loops, and for all Java code, so this limitation of the parallel loop semantics is not worse than the limitations of the exception semantics for any other construct of Java.

Chapter 4

Examples

4.1 Video Rendering

Suppose we wished to use a computer to render frames of an animation. Because each frame is independent of other frames in the video, they can be rendered in parallel, saving computation time. A procedure to do this is:

```
void render_frames(Frame[] frames,int start,int end) {
    parfor (int i=start; i<end; i++) {
        images[i] = frames[i].render();
    }
}
```

This loop would render the separate frames of the sequence in parallel. However, if an exception occurred while rendering the frames, the loop would stop. Because many frames have already been rendered at this point, and because rendering is a computationally expensive task, we would prefer to keep the partial results even in the event of an exception, and use these when recovering from the exception. For example, if the machine runs out of memory while rendering the frames, we would like to be able to recover from this situation by saving the completed frames to disk, and then continuing where the program left off. It is advantageous to save a contiguous block of frames together because this maximizes the effectiveness of a compression algorithm. Our proposed exception semantics allows this to be expressed easily:

```
void render_frames(Image images[],Frame[] frames,int start,int end) {
    parfor (int i=start; i<end; i++) {
        images[i] = frames[i].render();
    } catch (OutOfMemoryException e) {
        //  Delete frames that come after the exception
        for (int j=i; j<end; j++) {
            images[j] = null;
        }
        //  Save earlier, completed images
        save_images(images,start,i-1);
        //  Delete completed frames to recover memory
        for (int j=0; j<i; j++) {
            images[j] = null;
            frames[j] = null;
        }
        //  Render remaining images through recursive call
        render_frames(images,frames,i,end);
    }
}
```

This example recovers from an exception by deleting all rendered images that come logically after the frame that caused the exception, and saving all images that come logically before the exception. Images that come logically after the exception are not saved, because they may not be complete, and would not be contiguous with the earlier frames. Because the postcondition guarantees that the exception thrown is the structurally first exception and that all prior iterations have completed, these images can be saved to disk together and do not need to be recomputed. Because these images are in a contiguous block without holes, it should be possible to compress them easily; this would not be possible if all completed images were saved, including any completed frames logically after the image that caused the exception.

4.2 Try..Repair..Retry Strategy

Heinz94 demonstrates exception semantics for Modula-3* using a simple example of how to use the try..repair..retry paradigm to handle exceptions in parallel loops. The example uses a `parfor` loop which calls a `ComputeData` function on each item in an array. If an exception occurs in one iteration, the program attempts to repair the data and attempt to process it again; if even this fails, the `BadData` exception is propagated out of the loop, a global repair on the entire array is attempted, and the program tries again. The example does not define the compute and repair procedures, but is intended to show how their parallel exception semantics allows the expression of this try..repair..retry strategy. The example was written in Modula-3*, but can be translated directly to Java:

```
void ProcessData(int k) throws BadData {
     parfor (int i=1; i<=k; i++) {
          try {
               ComputeData(i); // first local trial
          } catch (BadData ex) {
               LocalDataRepair(i); // local repair
               try {
                    ComputeData(i); // second local trial
               } catch (Exception any) {
                    // any exception implies data still bad
                    throw new BadData();
               }
          } catch (Exception any) {
               // prevent the propagation of all other exceptions
          }
     }
}

void HandleData() throws Exception {
     int n;
     while (true) {
          n = InputData(); // get the next data to be processed
          try {
               ProcessData(n); // first global trial
          } catch (BadData ex) {
               GlobalDataRepair(n); // global repair
               ProcessData(n); // second global trial
          }
     }
}
```

Because the parallel exception model proposed by Heinz does not provide any guarantees about which iterations have completed after an exception has been thrown, the `GlobalDataRepair` function and the second call to `ProcessData` cannot use this information, and may unnecessarily re-process data that has already been processed correctly. In addition, because the semantics does not allow multiple inconsistent exceptions to be thrown from the same loop, the `ProcessData` function must be written in such a way that only one kind of exception, `BadData` can be thrown. Additional exception types could not be handled simply by inserting additional handlers for them; this would require completely restructuring the code.

This example can be simplified using the structured exception semantics:

```
void ProcessData(int start,int end) {
     parfor (int i=start; i<=end; i++) {
          try {
               ComputeData(i); // First local trial
          } catch (BadData ex) {
               LocalDataRepair(i); // Local repair
```

```
                ComputeData(i); // Second local trial
            }
        } catch (BadData ex) {
            GlobalDataRepair(i,end); // Global repair of i..end
            ComputeData(i); // Re-process offending iteration
            ProcessData(i+1,end); // Re-process remaining iterations
        }
    }

    void HandleData() {
        while (true) {
            n = InputData();
            ProcessData(1,n);
        }
    }
```

In this version of the example, the global exception handler attempts to repair and re-process only the data that needs it. This is possible because the exception semantics guarantee that all iterations structurally prior to the iteration that generates the propagated exception complete normally, and because the `for..catch` construct allows the index of the iteration that generated the exception to be used in the handler.

There is no need for extra code to make sure that only `BadData` exceptions are thrown, because the exception semantics allow more than one exception to be thrown within the same loop without introducing a consistency problem. If more kinds of exceptions than `BadData` needed to be handled, this could be done by simply adding extra `catch` clauses for the new exceptions, either at the local or global level, without completely restructuring the code.

The exception handler attempts to recompute the iteration that generated the exception separately before the call to `ProcessData` that computes the remaining portion of the data. If the attempt to recompute the iteration that caused the exception fails, then this means that the `GlobalDataRepair` failed, and the program should stop rather than attempt to recompute the remaining data. If the recomputation of the iteration that generated the exception was not separated from the recursive call to `ProcessData`, this could result in an infinite loop.

Chapter 5

Implementation

The proposed language extensions can be implemented using a library of new classes that encapsulate the loop and exception semantics, and a source-to-source translator that translates the extensions into standard Java code that uses the library. The resulting code can then be compiled by any Java compiler. This chapter describes the implementation of the library, and the design of a source-to-source translator. The source-to-source translator for our extensions has not yet been implemented, but similar extensions have been implemented as source-to-source translators before, and implementation of our extensions would be straightforward.

5.1 Strategy

An extension to Java could be implemented as a Java library, a source-to-source translator, a full compiler generating standard JVM bytecode, or a compiler generating special bytecode for a modified JVM.

A modified JVM would allow the most flexibility in what can be implemented, but would come at the cost of portability and interoperability with existing Java code. The JVM's built-in `Thread` class could be redefined so that uncaught exceptions generated by the thread would be passed to the parent thread upon termination; the parent thread could receive the exceptions when it attempts to `join` the child threads. This implementation technique would require a compiler to recognize the `parfor` loops and generate the code needed to implement them.

A full compiler has the ability to perform some transformations that generate valid JVM bytecode but cannot be expressed in Java in source form, but is also complex and can introduce interoperability problems. A source-to-source translator cannot perform as many transformations as a full compiler, but because its output is standard Java source code, it does not introduce any portability or interoperability problems. In addition, the code generated by a source-to-source translator is readable, which makes debugging and analysis easier. For these reasons, we chose to implement the extensions as a library along with a description of a source-to-source translation. A more detailed analysis of these options for implementing parallel loops in Java is provided by Philippsen [16].

The implementation described here uses an inner class to represent each parallel loop, and a set of methods that spawn threads, execute the loop, and wait for the threads to terminate. If an exception is generated by the loop, the loop class waits for structurally prior iterations to complete, and re-throws the structurally first exception to the calling code. This allows the parallel loop constructs to be implemented without changing the virtual machine, and allows code with parallel loops to work transparently with standard Java code.

The library described here uses the simple approach of spawning a separate Java thread to execute each iteration of the body of the loop. Other methods of implementing parallel loops in Java achieve higher performance by not using a one-to-one mapping of loop iterations to threads; see [19] for a comparison. However, such a strategy must deal with the possibility that there are dependencies between different iterations, and it can be difficult to determine when two iterations can be executed sequentially and when they must be executed concurrently. The mechanisms needed to deal with this situation are complex. Because our purpose here is to demonstrate our proposed exception semantics, rather than how to achieve optimum performance, we chose the simpler one-to-one mapping of iterations to threads.

Although we have not implemented an automatic source-to-source translator for our proposed extensions, we describe here how the extended features could be translated mechanically into normal Java code. Other researchers have implemented automatic translators that generate standard Java code for parallel loops using similar techniques [17] [16] [19] [20], and it should be straightforward to modify one of these translators to use our proposed exception semantics.

5.2 Loop Translation

To implement `parfor` loops, the `ParforLoop` class manages a single parallel loop and is responsible for creating threads, waiting for them to complete, catching exceptions, interrupting running threads, and propagating exceptions back to the caller. `ParforLoop` has undefined abstract methods for the condition expression, update statement, and loop body; each method takes an `Object` parameter that holds the current value of the loop's iteration variable. The class is not instantiated directly: in order to use the class, a child class must be derived from it that defines these methods for the particular loop that is to be executed.

A `parfor` loop is translated as an inner class that extends the `ParforLoop` class and defines the abstract methods, and executed by calling the ParforLoop class's `loop` method. This transformation requires a few fairly simple substitutions. The generic loop,

```
parfor (type var = initial_value; condition_expr; update_stmt) {
     loop body
} catch (exception_type e) {
     exception handler
} finally {
     finally clause
}
```

would be translated as:

```
{
     class ParforLoop_0 extends ParforLoop {
          boolean condition(Object index) {
               type var = (type) index;
               return condition_expr;
          }
          Object update(Object index) {
               type var = (type) index;
               update_stmt;
               return var;
          }
          void body(Object index) {
               type var = (type) index;
               loop body;
          }
     }
     ParforLoop_0 loop = new ParforLoop_0();
     type var = initial_value;
     try {
          loop.loop(var);
     } catch (exception_type e) {
          var = (type) loop.getIteration();
```

```
            exception handler
        } catch (RuntimeException rte) {
            throw rte;
        } catch (Throwable t) {
            throw new UnexpectedLoopException(t);
        } finally {
            var = (type) loop.getExceptionIndex();
            finally clause
        }
    }
```

The loop is described by overriding the abstract `condition`, `update`, and `body` functions to contain the loop's condition expression, update statement, and body implementation, respectively. These functions each take a reference to the iteration variable as a parameter. The loop is actually executed by the call to the `ParforLoop.loop` method, which is passed the initial value of the iteration variable. This method executes the control portion of the loop and starts threads to execute the body of the loop, and does not return until the loop has terminated. If the loop terminates abnormally, the `loop` method propagates the structurally first exception.

If the loop has any extended `catch` or `finally` clauses attached, for which the iteration variable is to remain in scope, the handlers described there are attached to the `try..catch` block surrounding the call to `loop`. The value of the iteration variable is retrieved by calling the `getIteration()` method of the `ParforLoop` class.

Because the `loop` method calls user-defined code which may throw exceptions, and because all checked exceptions must be declared in Java, the `loop` method is declared to throw `Throwable`, the base class of all exceptions in Java. It would not be acceptable to require all methods that use parallel loops to throw `Throwable`, however, so the `try..catch` block is used to catch all exceptions from the loop. Exceptions that could legitimately be thrown by the loop must be caught and re-thrown, while a final `catch(Throwable)` clause is added to catch any other exception and prevent `Throwable` from becoming a required part of the enclosing method's exception signature. The translator must analyze the loop's code to determine what checked exceptions could be thrown by the loop, and generate a `catch` clause that simply re-throws the exception for each case. If any exception that could not be legitimately thrown by the loop is caught, this indicates a implementation error, and is handled by raising an UnexpectedLoopException; if the translation is done correctly, this should never execute.

In order to allow an iteration variable of any type to be used, the variable is passed as an Object reference, and must be cast to a variable of the appropriate type before being used. The update method must make a copy of the iteration variable and return the copy, rather than modifying the iteration variable directly, in order to prevent it from interfering with the loop body, which runs concurrently and also has a reference to the iteration variable. This copying causes each iteration of the loop body to have its own copy of the iteration variable, so that they can run concurrently.

5.3 Scope Issues

Because the loop body, condition expression, and update statement are defined inside an inner class, final variables from the method in which the loop is defined are still in scope, and can be used in the definitions. Non-final variables cannot be accessed from inside an inner class, because Java does not allow references to objects on the stack to be constructed, so any shared variables must be either declared final or stored as objects on the heap. Member variables and methods of the enclosing class also remain in scope, and can be used without restriction, provided that proper synchronization is used to prevent simultaneous access and modification of shared variables.

In order to allow access to non-final stack variables, these variables must be moved to the heap, because Java does not allow multiple threads to share the same stack. Although the JVM provides no way to create references to objects that already exist on the stack [21], it is possible to copy references to objects on the heap, or to create handles (references to references) to objects on the heap. A translator could do this by identifying which stack variables will by accessed by the parallel loop code, and replacing any non-final variables with handles to objects on the heap. A simple way to do this is to replace each stack variable with a reference to an array of size 1, and replace each access of the variable with an access to the element of the array. Because arrays are objects stored on the heap, this transformation allows the thread to access and even assign to the variable in question.

The following simple loop access a non-final local variable as it sums an array:

```
int sum_array(int A[]) {
    int sum;
    sum = 0;

    parfor (int i; i<A.length; i++) {
        synchronized(this) {
            sum += A[i];
        }
    }
```

```
        return sum;
    }
```

The loop could be transformed as follows to allow access to the local variable from the body of the loop:

```
    int sum_array(int A[]) {
        int sum[] = new int[1];
        sum[0] = 0;

        parfor (int i; i<A.length; i++) {
            synchronized(this) {
                sum[0] += A[i];
            }
        }

        return sum[0];
    }
```

By storing `sum` in an array, the variable is moved from the stack to the heap, where it can be accessed by the inner class that will implement the `parfor` loop.

5.4 The ParforLoop Library

The main class of the library is the `ParforLoop` class. It is an abstract class which has undefined methods for `condition`, `update`, and `body`, which are used to describe the work to be done by the loop. The loop is actually executed by the `loop` method of the class. The class also maintains bookkeeping information about the loop, the iteration variable, any exceptions generated by the loop, and references to the threads which execute the iterations of the loop.

The `parfor` semantics are implemented by the `loop` method. This method is divided into two parts: the control portion of the loop which spawns threads to execute the iterations of the body, and a cleanup portion of the loop which waits for spawned threads to complete and re-throws any generated exception.

The control portion of the loop uses the abstract `condition` and `update` methods, which specify what particular loop is to be executed. It executes the loop control sequentially, by calling the `condition` and `update` methods on the iteration variable until the condition is false. For each iteration, it creates and starts a `StructuredThread` object to execute the body for that iteration of the loop concurrently. The `StructuredThread` objects are stored in a Java `Vector` so that they may be referred to later in the cleanup phase of the loop. If an exception occurs, or if the loop itself is interrupted, the loop control terminates without creating any further threads. The test to see if any loop iteration has terminated exceptionally and the task of starting a new thread must occur atomically to prevent a race condition that could occur if a child thread terminated exceptionally and attempted to interrupt running threads at the same time that the loop control modified list of running threads.

```
for (iteration = 0;
     !exception_occurred && !Thread.currentThread().isInterrupted();
     iteration++) {

     boolean loop_test=false;
     try {
          loop_test=condition(index);
     } catch (Throwable exception) {
          catchException(iteration,index,exception);
          break;
     }

     if (!loop_test)
          break;

     synchronized(this) {
          if (exception_occurred)
               break;
          StructuredThread thread =
               new StructuredThread(this,iteration,index);
          threads.add(thread);
          thread.start();
     }

     try {
          index = update(index);
     } catch (Throwable exception) {
          catchException(iteration,index,exception);
     }
}
```

After all threads have been started, the loop enters the cleanup phase. The implementation loops over all iteration threads and waits for each iteration to terminate by calling the `join` method on each thread. After the loop has terminated, it checks to see if an exception has been thrown, and if one has, it re-throws it; otherwise, it terminates normally. If the parent thread of the loop is interrupted, this interruption is passed down to all remaining child threads and reasserted at loop termination. This allows an interruption to cancel an entire subtree of threads safely, which is necessary if `parfor` loops are nested.

```
boolean interrupt_occurred=false;
```

```
        for (iteration=0; iteration < threads.size(); ) {
            try {
                StructuredThread thread =
                        (StructuredThread) threads.get(iteration);
                thread.join();
                iteration++;
            } catch (InterruptedException ie) {
                if (!interrupt_occurred) {
                    interruptLaterThreads(iteration);
                    interrupt_occurred=true;
                }
            }
        }

        if (interrupt_occurred) {
            Thread.currentThread().interrupt();
        }

        if (exception_occurred) {
            throw exception_object;
        }
```

The `StructuredThread` class extends `Thread` and executes one iteration of the loop by calling `ParforLoop.body` with that iteration's copy of the loop variable. If that iteration terminates with an exception, it notifies the `ParforLoop` object of the exception by calling `catchException`, and terminates.

The exception semantics is implemented primarily by the `ParforLoop.catchException` method. This method determines if the exception passed to it is the logically first exception that has been generated by the loop so far. If it is, it stores the exception, a copy of the iteration variable for the iteration that generated it, and interrupts all logically higher iterations. This allows any logically prior iterations to continue running, and any of them may themselves call `catchException`. Because `catchException` only takes action when the exception passed to it is the logically first exception of the ones thrown so far, it follows that when all iterations of the loop have completed, the exception stored in the `ParforLoop` object must be the logically first exception of the loop. This value can then be re-thrown by the `loop` method after it has joined all threads in the loop.

```
    void catchException(int iteration,
                        Object index,
                        Throwable exception) {
        synchronized (this) {
            if (!exception_occurred || iteration < exception_iteration) {
                exception_occurred = true;
                exception_iteration = iteration;
                exception_index = index;
                exception_object = exception;
                interruptLaterThreads(iteration+1);
            }
        }
    }
```

5.5 Hierarchical Thread Termination

When an interruption occurs, the thread that receives the interruption should terminate safely and promptly. When the parent thread of a parallel loop receives an interruption, the loop must terminate, and this is done by propagating the interruption downwards to the loop iteration threads, causing the entire tree of threads to terminate. Our semantics still require that even in the case of an exception or interruption, the loop must not terminate and return control to the caller until all iterations within the loop have completed.

To accomplish these goals, the `loop` method is written so that if it receives an interruption while creating the child threads, it stops creating new threads and proceeds directly to the loop that joins the loop iteration threads without clearing the interruption. The loop that joins the child threads detects interruptions and responds to them by sending an interruption to each child thread that has not yet been joined, and continues to join child threads until the loop is complete. Upon termination, it re-asserts the interrupt status of the parent thread, and if a child thread terminated with an exception, it re-throws the structurally first exception. This strategy propagates interruptions downwards to child threads while allowing exceptions to propagate upwards to parent threads, and preserves the requirement that all child threads must terminate before the loop as a whole terminates.

Because it is possible to nest parallel loops, it is possible for the parent thread of an inner loop to be interrupted because an exception occurred in another thread in the outer loop. The interruption is propagated downwards to the iterations of the inner loop, and after all of them have terminated, control is returned to the parent thread, which can then terminate. If one of the iterations of the inner loop terminates with an exception, this exception is still propagated upwards to the outer loop.

It is important to distinguish between thread interruption and the `InterruptedException`. A thread can receive an interruption at any time, and when it does, it continues to execute normally. The interrupt status of the thread must be tested for and reset by the thread itself. Some methods, such as `Thread.sleep` or `Thread.join` that can cause a thread to block will test for thread interruption and wake the thread up by throwing an `InterruptedException` to indicate that the method was unable to complete because of the interruption. This allows interrupted threads to terminate promptly even if they make blocking calls.

Because of the differences between a thread interruption and the `InterruptedException`, they are handled separately. An `InterruptedException` is propagated upwards to the parent thread like any other exception. Because it is consistent for a thread to be interrupted and to throw an exception simultaneously, at the termination of the loop, the `loop` method tests for interruption of the loop control and exceptions from loop iterations and propagates them independently. If a child thread makes a blocking call and does not handle the `InterruptedException` that results if the thread is interrupted, that exception may propagate to the parent thread, and in this case the loop will both propagate the `InterruptedException` and leave the thread in an interrupted state.

5.6 Performance

The overhead associated with our exception semantics should not significantly affect the performance of concurrent Java programs. The cost of setting up the loop and starting threads is linear in the number of iterations of the loop. In the normal case in which there are no exceptions, the cost of joining the threads in the loop and cleaning up is also linear in the number of iterations of the loop, and introduces little overhead.

If the loop terminates exceptionally, the running time may be longer than other methods because our semantics requires all earlier iterations of the loop to complete when an exception is thrown, while other semantics allow all threads to be terminated immediately in this situation. However, the running time of this should not be any worse than that of normal execution of the loop. It is possible that a loop which uses concurrency control constructs incorrectly or which ignores the `InterruptedException` may deadlock or enter an infinite loop if an exception occurs; however, this is the result of an incorrect program and not the result of the translation.

The translation does increase code size. Any program using the translation must use the `ParforLoop` library. The translation itself creates a new inner class and several methods for each loop, and creates additional code to handle each type of exception that could be expected from the loop. The increase in code size is linear in the number of translated `parfor` loops. Relaxation of the Java requirement that all exceptions must be caught or declared (except for errors and run-time exceptions) would make the translation simpler and would reduce the size of the translated code.

Chapter 6

Conclusions

The goal of our semantics for exceptions in concurrent loops is to facilitate the writing of concurrent programs by allowing exceptions to be used in a consistent way with other language features, while allowing us to reason about the program's behavior in the event of an exception. In this chapter, we examine how the proposed semantics meets this goal, and what directions this research might take us in the future.

6.1 Evaluation

The way that exceptions behave and are used in concurrent programs should be a consistent extension of the way that exceptions are used in sequential code. Our semantics does this by allowing exceptions to be used in concurrent code and allowing uncaught exceptions to be propagated up the call chain to matching exception handlers in the same manner that sequential exceptions are handled. Our semantics for exceptions in concurrent loops is an extension of the semantics for sequential loops, and by canceling structurally later iterations while allowing earlier iterations to complete in the event of an exception, it is analogous to the behavior of a sequential loop in the event of an exception.

In order to make concurrent programs predictable and understandable, it is important that the effect of a program is determined by the structure of the program, rather than the accident of how it is executed. Although nondeterminism is not always harmful and cannot be eliminated from concurrent programs, it can be tamed by using constructs that allow the program to proceed in a concurrent way without making the effect of the program unpredictable. Our semantics meets this goal because the result of a loop that terminates with exceptions is determined by the way in which the loop was written rather than the order in which the exceptions occur. Because the structurally first exception is propagated out of the loop, and earlier iterations are allowed to complete, loops can be written so that they will terminate in a consistent and predictable way even if multiple exceptions occur. This predictability can be especially important when debugging concurrent programs, because it allows some classes of bugs to be reproduced.

Because it is important to be able to reason about a program's behavior in order to write correct programs, it is important to be able to make strong assertions about a program's behavior even in the event of an exception. Our semantics allows meaningful postconditions to be asserted about concurrent loops that terminate exceptionally, because it provides a guarantee that iterations prior to the iteration that generated the exception have completed normally, and an exception handler can correctly assume this and use any partial results generated by the loop in its recovery. Our semantics also makes it possible to write concurrent loops that are guaranteed to terminate in the event of an exception, even if there are dependencies between iterations of the loop, provided that there are no forward dependencies. This allows concurrent programs to be written that have well-defined behavior even in exceptional conditions.

The proposed translation of the extensions shows that the semantics can be implemented in Java with modest overhead and without modifying the Java virtual machine. Other strategies for translation that do not use a one-to-one mapping of iterations to threads may achieve higher performance and less overhead. Cancellation of running iterations once an exception has occurred can improve performance in the case that the results of those iterations are not useful given the exception, but prior results are useful.

The semantics presented here should be useful for scientific computing and for applications where parallelism in the program is used to improve performance, and where partial results are useful. It may be useful for debugging these kinds of applications, even where partial results are not useful, because it allows determinism in the case of multiple exceptions. It may also be useful in libraries where exceptions are caused not by programming errors in the library but bad data passed to the library, because it integrates the exception semantics with the concurrency constructs, so a programmer need not know whether a method in a library uses parallel loops in order to write correct code that uses exceptions.

Our semantics is most likely not useful for systems programming or real-time systems where concurrency is an inherent part of the problem to be solved, rather than a means to better performance. It is generally not useful for inherently concurrent algorithms, or for separate, unrelated concurrent tasks such as maintaining a GUI and performing a computation in the background. These kinds of computations do not benefit from parallel loops, and are better expressed as separate routines executing concurrently rather than one loop executing in parallel. The kind of concurrency normally provided by Java Threads [4] or Ada tasks [22] is probably more appropriate for this kind of system.

6.2 Future Work

For some loops, it does not matter which exception is structurally first, or what partial results have been computed, and for these loops, propagating the first exception in time and canceling or interrupting all remaining threads in the loop may make more sense. Because it may even be desirable to use both semantics within the same program, one possible approach is to one keyword to indicate parallel loops for which partial results are useful and deterministic exception semantics is desired, in which our structured exception semantics would be used, and another keyword to indicate parallel loops which express inherently concurrent algorithms for which it is not important which exception is structurally first.

Although our proposal is made in the context of concurrent loops, a similar semantics may be useful for other concurrent constructs and could achieve many of the same goals. A structured exception semantics for a parallel block could propagate the structurally first exception from the block while interrupting later statements in the block. Termination conditions and postconditions for the block could be constructed which are similar to the conditions for the concurrent loop, except without reference to an iteration variable.

It is possible that a similar technique could be used to implement the `break` and `return` statements inside parallel loops, so that everything that can be done within a sequential loop can be done in a parallel loop. The `break` keyword could be defined to interrupt or cancel all structurally later iterations in the loop, while allowing prior iterations to continue. Similarly, the `return` keyword could be defined such that value returned by the structurally first `return` statement that is executed in the loop is the value that is returned by the method. This could be useful, for example, in a function that searches an array in parallel for the first member that meets some criteria and returns it, especially if the test is a computationally intensive one. Further study would be needed to determine if such a semantics would be useful or would make programs more understandable.

Our semantics is presented in the context of Java, but it could be applied to other languages. The partial results feature may be particularly helpful in implementing the `retry` keyword in languages that have one, such as Eiffel [23]. This would allow the loop to restart at the point the logically first exception occurred, while keeping previously computed partial results, and should greatly simplify the `try`..`repair`..`retry` pattern described in the examples.

The ability to express concurrency in a clean and natural way is an important feature to have in modern programming languages. Parts of programs that express concurrency or parallelism should be well integrated with the rest of the language, and it is important for language designers to consider how concurrency interacts with exception handling in particular. We have presented a concurrency construct for Java that provides exception semantics that are consistent and integrated with the language, and that can be implemented with modest overhead.

Appendix A

ParforLoop.java

```
import java.util.*;

public abstract class ParforLoop {
    Vector /* of StructuredThread */ threads;

    boolean   exception_occurred  = false;
    Throwable exception_object    = null;
    int       exception_iteration = 0;
    Object    exception_index     = null;

    String name = "";
    static int loop_count = 0;

    public ParforLoop() {
        synchronized(ParforLoop.class) {
            name = "loop"+loop_count;
            loop_count++;
        }
    }

    public ParforLoop(String name) {
        this.name=name;
    }

    public void loop(Object initial_value) throws Throwable {
        int iteration;
        Object index = initial_value;
        threads = new Vector();

        for (iteration = 0;
             !exception_occurred && !Thread.currentThread().isInterrupted();
             iteration++) {
```

```
            boolean loop_test=false;
            try {
                loop_test=condition(index);
            } catch (Throwable exception) {
                catchException(iteration,index,exception);
                break;
            }

            if (!loop_test)
                break;

            synchronized(this) {
                if (exception_occurred)
                    break;
                StructuredThread thread =
                    new StructuredThread(this,iteration,index);
                threads.add(thread);
                thread.start();
            }

            try {
                index = update(index);
            } catch (Throwable exception) {
                catchException(iteration,index,exception);
            }
        }

        boolean interrupt_occurred=false;

        for (iteration=0; iteration < threads.size(); ) {
            try {
                StructuredThread thread =
                        (StructuredThread) threads.get(iteration);
                thread.join();
                iteration++;
            } catch (InterruptedException ie) {
                if (!interrupt_occurred) {
                    interruptLaterThreads(iteration);
                    interrupt_occurred=true;
                }
            }
        }

        if (interrupt_occurred) {
            Thread.currentThread().interrupt();
        }

        if (exception_occurred) {
            throw exception_object;
        }
    }
    void catchException(int iteration,
                        Object index,
                        Throwable exception) {
        synchronized (this) {
            if (!exception_occurred || iteration < exception_iteration) {
                exception_occurred = true;
                exception_iteration = iteration;
                exception_index = index;
                exception_object = exception;
                interruptLaterThreads(iteration+1);
            }
        }
    }

    void interruptLaterThreads(int start) {
        for (int i=start; i < threads.size(); i++) {
            StructuredThread thread = (StructuredThread) threads.get(i);
            thread.interrupt();
        }
    }

    public Object getExceptionIndex() {
        return exception_index;
    }

    abstract boolean condition(Object index) throws Throwable;
    abstract Object update(Object index) throws Throwable;
    abstract void body(Object index) throws Throwable;
};

class StructuredThread extends Thread {
    ParforLoop loop;
    int iteration;
    Object index;

    StructuredThread(ParforLoop loop,int iteration,Object index) {
        this.loop = loop;
        this.iteration = iteration;
        this.index = index;
    }

    public void run() {
        try {
            loop.body(index);
        } catch (Throwable exception) {
            loop.catchException(iteration,index,exception);
        }
    }
}
```

```
class UnexpectedLoopException extends RuntimeException {
    final Throwable exception_object;
    public UnexpectedLoopException(Throwable t) {
        exception_object = t;
    }
    public Throwable getException() {
        return exception_object;
    }
    public String toString() {
        return "UnexpectedLoopException: "+exception_object.toString();
    }
}
```

Bibliography

[1] American National Standards Institute, Inc. *The Programming Language Ada Reference Manual*. Springer-Verlag, 1983.

[2] F. Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the Winter 1993 USENIX Technical Conference and Exhibition*, pages 29–41, San Diego, CA, 1993.

[3] D. A. Solomon. *Inside Windows NT*. Microsoft Press, Redmond, WA, second edition, 1998.

[4] Sun Microsystems. *Java 2 Platform, Standard Edition, v 1.2.2 API Specification*, 1999. http://java.sun.com/products/jdk/1.2/docs/api/index.html.

[5] P. Carlin, K. M. Chandy, and C. Kesselman. The Compositional C++ language definition. Technical Report 1993.cs-tr-92-02, Department of Computer Science, California Institute of Technology, 12, 1993.

[6] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.

[7] A. S. Tanenbaum. A tutorial on Algol 68. *Computing Surveys*, 8(2):155–190, June 1976.

[8] D. Stoutamire and S. Omohundro. The pSather 1.1 manual and specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, CA, 1996.

[9] E.A. Heinz. Modula-3*: An efficiently compilable extension of modula-3 for explicitly parallel problem-oriented programming. In *Joint Symposium on Parallel Processing*, pages 269–276, Tokyo, May 1993. Waseda University.

[10] J. C. Adams. *Fortran 95 Handbook*. MIT Press, Cambridge, MA, 1997.

[11] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, second edition, 1997.

[12] J. D. Ichbiah, J. C. Heliard, O. Roubine, J. G. P. Barnes, B. Krieg-Brueckner, and B. A. Wichmann. Rationale for the design of the ADA programming language. *ACM SIGPLAN Notices*, 14(6):1–247, 1979.

[13] C. Fleiner, J. Feldman, and D. Stoutamire. Killing threads considered dangerous, 1996.

[14] S. Murer, J. A. Feldman, C. Lim, and M. Seidel. pSather: Layered extensions to an object-oriented language for efficient parallel computation. Technical Report TR-93-028, International Computer Science Institute, Berkeley, CA, 1993.

[15] E. A. Heinz. Sequential and parallel exception handling in Modula-3*. In P. Schulthess, editor, *Advances in Modular Languages: Proceedings of the Joint Modular Languages Conference*, pages 31–49, Ulm, Germany, Sep 1994.

[16] M. Philippsen. Data parallelism in Java. In J. Schaefer, editor, *High Performance Computing Systems and Applications*, pages 85–99. Kluwer Academic Publishers, Boston, Dordrecht, London, 1998.

[17] B. Blount, S. Chatterjee, and M. Philippsen. Irregular parallel algorithms in JAVA. In *Parallel and Distributed Processing, 6th International Workshop on Solving Irregularly Structured Problems in Parallel*, number 1586 in Lecture Notes in Computer Science, pages 1026–1035, Puerto Rico, April 16, 1999. Springer Verlag.

[18] Sun Microsystems. *Java 2 Platform, Standard Edition, v 1.2.2 API Specification*, 1999. http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html.

[19] J. Oliver, E. Ayguade, and N. Navarro. Towards an efficient exploitation of loop-level parallelism in Java. In *Java Grande*, pages 9–15, 2000.

[20] C. van Reeuwijk, A.J.M. van Gemund, and H.J. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, November 1997.

[21] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1997.

[22] J. G. P. Barnes. An overview of Ada. *Software - Practice and Experience*, 10(11):851–887, 1980.

[23] B. Meyer. *Eiffel: The Language*. Prentice Hall International, Hemel Hempstead, UK, 1992.