# CC++: A Declarative Concurrent Object Oriented Programming Notation

K. Mani Chandy Carl Kesselman
California Institute of Technology

March 12, 1993

**Abstract**

CC++ is Compositional C++, a parallel object-oriented notation that consists of C++ with six extensions. The goals of the CC++ project are to provide a theory, notation and tools for developing reliable scalable concurrent program libraries, and to provide a framework for unifying:

1. distributed reactive systems, batch-oriented numeric and symbolic applications, and user-interface systems,
2. declarative programs and object-oriented imperative programs, and
3. deterministic and nondeterministic programs.

This paper is a brief description of the motivation for CC++, the extensions to C++, a few examples of CC++ programs with reasoning about their correctness, and an evaluation of CC++ in the context of other research on concurrent computation. A short description of C++ is provided.

## 1 Introduction

Compositional C++, or CC++ is a concurrent object-oriented programming notation. CC++ is C++ with six extensions for writing declarative and concurrent programs. The declarative extensions are particularly suited for

rapid prototyping. These extensions are used to *compose* programs to execute on distributed environments; the emphasis in compositional programming is on methods for putting programs together to execute concurrently so that the specification for a composed parallel program can be proved, in a simple way, from specifications of its components.

CC++ is a notation for reactive systems executing on heterogeneous distributed environments as well as for applications executing on parallel supercomputers.

The extensions of CC++ are based, in part, on ideas from Concurrent Logic Programming [Sha86], Strand [FT89], PCN [CT91, FT92], and dataflow languages [Ack82]. Methods for reasoning about correctness are based on UNITY [CM88].

**Organization of the Paper** The next section presents the motivation for CC++. Since CC++ consists of a small number of extensions to C++, some understanding of C++ is helpful in reading this paper; therefore, a very brief overview of C++ is provided in Section 3, for those unfamiliar with it. Section 4 describes the extensions in CC++, and Section 5 presents examples of parallel programs with methods for reasoning about their correctness. A description of an implementation of CC++ is given in Section 6. Section 7 compares CC++ with other object-oriented notations. Section 8 is a summary of the main points of the paper.

# 2 Motivation

The motivation for CC++ is given in this section. We conclude each topic in this section with a line, in *emphasis* font stating the relevance of each issue to CC++ design.

## 2.1 Integrating Different Approaches to Concurrency

**Integration of Parallel and Reactive Systems** Within a decade, parallel supercomputers will be integrated into reactive systems such as command and control systems, communication systems, process-control systems, and perhaps even automobiles. The same program may deal with differ-

ential equations, receive data from distributed sensors, control distributed processes, and interact with people.

Such a system has three main parts:

1. distributed control,
2. core numeric and symbolic computing,
3. user interfaces.

Parallel supercomputing usually deals with only the second of the three parts.

User-interface systems are thought of as sequential programs in which an event-manager schedules many objects such as windows and queues. Treating such systems as concurrent systems, with concurrently executing objects, simplifies design even if the computing platform is a sequential machine.

A common thread that unites all three parts of reactive systems is concurrency. Therefore, *a unified framework should be provided to manage concurrency in distributed control, core numeric and symbolic computation, and in user interfaces.*

**Integration of Data Parallel and Asynchronous Computation** Data parallelism has been shown to be a powerful model of computation for scientific computing. The data-parallel model allows programmers to treat a distributed object — such as an array distributed across several nodes of a multicomputer — as a single object. Programmers can think of their programs as sequential programs in which certain operations are implemented as concurrent operations. Data parallel programs are attractive because they allow programmers to execute sequential programs on parallel machines by using distributed objects.

By contrast, the model of computing in reactive systems is that of asynchronous communicating objects. Programmers do not have the luxury of thinking of their programs as sequential programs in which some operations are executed concurrently because reactive systems, by their very nature, have multiple independent threads of control.

Both the data parallel model and the asynchronous model of computation are useful. The integration of parallel supercomputing with reactive computing requires *notations and tools for integrating data parallel and asynchronous computation models.*

**Integration of Deterministic and Nondeterministic Programs** Most batch-oriented numeric and symbolic programs are deterministic. Determinism has several advantages, including repeatability of execution. Reactive programs, however, interact with their environment which is essentially unpredictable and uncontrollable. Therefore, reactive programs are nondeterministic in nature. The integration of parallel supercomputing with reactive computing requires *a unified framework that allows programmers to develop deterministic programs or nondeterministic programs, as needed. Demonstrating that a program is deterministic should be straightforward.*

**Nonuniform Memory Access** The time required to access data in a reactive system will vary depending on where the data is located. Though arguments can be made in favor of uniform memory access in parallel supercomputers, these arguments do not apply to distributed reactive systems. The integration of supercomputing with reactive systems requires that programmers pay attention to data locality in some cases (where data is geographically distributed). *The compiler, run-time system and to (some extent) the programmer must be cognizant of nonuniform access times of a distributed system.*

## 2.2 Developing Correct Concurrent Programs

**Demonstrating Correctness** Since debugging reactive programs is even more difficult than debugging sequential programs, *methods for developing reactive programs should include techniques for reasoning about program correctness.*

**Exception Handling and Debugging** One way to help in producing reliable code, is to provide mechanisms for inserting assertions (including invariants, and variant functions or metrics) into objects, and having the assertions checked at run-time. Run-time checks of assertions are particularly important in the development of libraries: the programmer who builds an object library can state the assertions to which users of the libraries must adhere, and run-time checks help protect the library builder from incorrect usage of the library [Mey88]. Violation of an assertion causes an exception that is then handled in some suitable way. *Exception handling mechanisms*

*are ideal vehicles for including assertions into objects.* Also, determinism in execution simplifies debugging — and CC++ programs can be written in a way that guarantees determinism.

**Strong Typing or No Typing?** Some concurrent notations, such as Strand [FT89], PCN [CT91], Concurrent Smalltalk [WHD89], and HAL [Hou92] have untyped variables. This has several advantages including the ability to use the same program for different types of arguments; for instance the same program can be used, without change, for lists of integers and lists of floating point numbers. An advantage of strong typing is that many errors can be detected at compile time. One reason for detecting errors of concurrent programs at compile time is that errors in concurrent systems can manifest themselves in strange ways such as starvation of some processes and these errors can be hard to detect and pinpoint at run-time. Programmers making the transition from sequential to concurrent programming find the development of concurrent programming to be difficult, and therefore, we think that *the advantages of strong typing outweigh its disadvantages.*

Some of the disadvantages of strong typing can be overcome by using C++ templates which are discussed later.

## 2.3 Software Engineering of Concurrent Programs

**Objects** Abstract data types, represented as objects, provide a clean interface between the use of the object and its implementation. An object can be implemented in different ways for different architectures, without changing the interface to the object. In particular, an object (such as an array) can be implemented in a distributed manner, and users of the object need not be concerned whether operations on the object are sequential or concurrent. Libraries of objects can be tailored, where necessary, for different kinds of architectures, without modifying programs that use the libraries. *An object-oriented language is an ideal vehicle for the development of concurrent programs.*

**Declarative Programs** In many cases, declarative programs are easier to develop and verify than imperative programs because declarative programs are closer to specifications. Therefore, declarative programs are well-suited

for rapid prototyping. One way to develop reliable programs is to develop a program in a declarative style and then transform those parts of the program that are used frequently into efficient imperative programs. *The integration of declarative and imperative programming supports step-wise development of programs.*

**Portability** Computing platforms of reactive systems will be diverse, including loosely-coupled networks of tightly-coupled shared-memory machines. Programs must be portable (to some degree) across the diversity of platforms to reduce the amount of effort required to port a program from one platform to another. Therefore, *constructs for concurrency should not be based on a particular class of machines.* For instance, the method by which processes synchronize should not depend on whether the hardware platform uses message-passing or shared memory, but should employ an abstraction of both mechanisms.

Language support should be provided for heterogeneous computing. In CC++ this support is provided by processor objects (or logical processors). There can be different classes of processor objects corresponding to different types of processors. The scheduler attempts to map a processor object on to the most appropriate physical processor. The semantics of a program are in terms of processor objects, and the implementation uses the semantic construct to obtain efficient implementations on heterogeneous networks.

**Templates** Some computational structures appear repeatedly in many different applications. Such computational structures can be represented as templates, and programs can be developed by "filling in" templates with different objects.

Examples of templates include parameterized objects in C++, higher-order functions in functional programming, and source-to-source program transformation notations in PCN [Fos91].

Templates can be re-used in different situations by suitably setting template parameters; thus template re-use improves productivity. *Use of template libraries will reduce program development time.*

**Transforming Sequential Programs** Many, though not all, parallel programs can be constructed by putting sequential programs together. *Methods*

*that help in constructing concurrent programs by modifying existing sequential programs, will remain useful.*

**New Language or Extension?** There are two basic approaches to designing a concurrent notation: propose a new notation or extend an existing sequential notation. The advantage of a new notation is that it can be kept simple. The disadvantages of a new notation are that programmers have to learn a new language to use concurrent computers, and a new set of tools have to be developed for the new notation.

Our goal is to help programmers use concurrent constructs quickly; the obstacle of learning a new notation, even a small notation, is overwhelming for many programmers. Therefore, we decided to implement a small extension to a sequential notation. To help programmers learn the extension quickly, and to allow easy adaptation of tools for the sequential notation to the concurrent notation, *the extension must be* **tiny** *and* **fit** *the base notation.* In particular, the extended notation must not restrict the base notation in any way: every program in the base notation must also be a program in the extended notation. Also, the programming paradigms of the base notation must be applicable in the extended notation, as well; for instance, if iteration is used in the base notation then the extended notation should allow iteration to be used to construct parallel programs as well.

**Single Paradigm for Concurrency or Many Paradigms?** One approach to developing parallel extensions to sequential languages is to insist that programmers use a single approach to obtain concurrency; the approach can be use of single-assignment variables, or monitors, or communicating processes, or functional programming. Another approach is to construct a *small* foundation that supports all of these approaches; if all these approaches for parallelism can be implemented on the small foundation, then programs using all these approaches are perforce programs written using the small foundation. We feel that a small foundation that supports a variety of parallel programming paradigms is useful to more programmers than a notation that only supports a single paradigm.

**Multilingual Programming** A reactive system may employ programs in many notations. For instance it may use Fortran for numeric computation,

Lisp or Prolog for symbolic computation, and C for process management. *Notations for concurrent programming should be able to call programs in other notations.*

## 2.4 Why CC++?

**Why Use C++ as a Base Language?** Our goal of helping many programmers migrate to concurrent computing suggests that we extend a widely-used notation. The goal of investigating the use of sequential libraries for concurrent program development suggests that we use a base language that has a substantial collection of libraries.

Our primary interest is not in the broad (and important) area of language design; we focus our interest on the narrow issue *of an extension to a sequential notation that helps in developing correct efficient concurrent programs.* Therefore, though our extensions should be simple, the underlying sequential notation can be arbitrary.

Sequential notations, that we could have considered for the base language include Fortran, C, C++, Cobol, Lisp, Pascal, Oberon and Smalltalk. Our goals of integrating symbolic and numeric programming; integrating user-interfaces, batch-oriented programs and reactive programs; providing support for multilingual programming; and investigating the potential re-use of sequential program libraries in concurrent programming suggested C or C++ in preference to the other notations. Our need for objects, templates, exception-handling, and type checking made C++ an obvious choice.

*Since C is a subset of C++, and since our extensions are compatible with C, we have, in effect, defined two extensions: Compositional C and Compositional C++.*

The central ideas of the extensions can be used with other languages that have user-defined data structures and classes.

Libraries have been developed recently in C++, for a variety of applications including numerics, combinatorics and user-interfaces. This supports the view that C++ can form a basis for integrating the development of the three parts of reactive systems: distributed control, core symbolic and numeric computation, and user-interfaces.

One of the disadvantages of C++ as a base language is that C++ is a

very rich notation: it has a wealth of features useful in a variety of situations. Giving the formal semantics for such a rich notation is difficult. A small notation with a complete axiomatic semantics is preferable for program verification. After weighing the tradeoffs, we decided to use C++, and reason about the correctness of *some* programs, rather than attempt to give proof rules that define the semantics of C++.

**Declarative Programming** A set of equations of the form:

$$(x = f(y)) \wedge (y = g(x))$$

where $f$ and $g$ are functions that satisfy certain properties, (and $\wedge$ is *logical and* ) can be executed to determine $x$ and $y$. The advantage of a program which is a set of equations is that in many cases, the specification is the program, and therefore the proof obligation — demonstrating that the program satisfies its specification — is straightforward.

Nondeterministic programs can be specified by a set of relations, such as: produce any $x$, $y$ that satisfy:

$$p(x \downarrow, y \uparrow) \wedge q(y \downarrow, x \uparrow)$$

where $p$ and $q$ are treated as both relations and procedures. The arrows $\downarrow$ and $\uparrow$ indicate data flow into or out of procedures; thus, given $x$ the procedure $p(x \downarrow, y \uparrow)$ computes any $y$ that satisfies relation $p$. For example, $x$ and $y$ may be lists where $y_0$ is a a constant; $y_{i+1}$ and $x_i$ satisfy some relation specified in $p$; and $x_i$ and $y_i$ satisfy some other relation specified in $q$ (and where the subscripts index the list).

Such a formula can be executed, with $p$ producing $y_i$, followed by $q$ producing $x_i$, for increasing $i$. Demonstrating correctness of a predicate calculus formula is often simpler than demonstrating correctness of an imperative program because the specification is often closer to the formula than to an imperative program.

Many specifications can be cast as conjunctions of implications and such specifications can also be transformed, in some cases almost syntactically, into declarative programs.

Our problem is to design a small declarative extension of C++ that allows programmers to write declarative programs such as these. The central design issue is to make the declarative extension a "natural" extension to C++.

The declarative extension is implemented by using data flow and single-assignment variables. Single assignment variables can be found in concurrent logic programming languages [Gre87, FT89], dataflow languages [Ack82, M+85] and others [CT91]. Single-assignment variables were proposed at least as early as the 1960s. A contribution of CC++ is to employ single-assignment variables in the context of a strongly-typed, object-oriented imperative language.

The effect of message-passing, semaphores, monitors [PBH77, Hoa74], barriers and `await` commands [OG76], can be obtained by using objects and sync variables. Thus programs that use these constructs can be used, with little change, in CC++.

# 3 A Brief Overview of C++

We present a brief overview of the C++ programming language. With a few exceptions, C++ is a pure extension of ANSI C. Most valid ANSI C programs are also valid C++ programs. C++ extends C by adding strong typing and language support for data abstraction and object oriented programming.

## 3.1 Strong Typing

Strong typing is essential to the construction of reliable programs. C++ has a more complete type-checking system than **C**. Function prototypes are required for all function definitions. A type-safe linkage mechanism is used to link together modules. Also, a type-safe dynamic data allocation mechanism is used which eliminates the need for implicit casting of `void` pointers. C++ includes a template facility that allows generic (or parameterized) functions to be defined in a type-safe manner. Experience has shown that when converting a program from C to C++, the type system of C++ uncovers errors at compile time that are undetected in the C program.

## 3.2 Data Abstraction and Encapsulation

C++ provides support for data abstraction and encapsulation by introducing the concept of a *class*, which is a generalization of a C structure. Classes are defined by the C++ keyword `class` as well as the C keywords: `struct`

and `union`. A class is like a C structure that can contain function members as well as data members. Thus for a class `C`, member access can take the form: `C.d` for data members or `C.f(a1,...,an)` for function members. The syntax for declaring a class is basically the syntax used to declare a structure in C.

A class object may reference itself by means of the `this` pointer. Within a member function, the compiler arranges for the keyword `this` to point to the object to which the member function belongs.

Program reliability is increased by ensuring that data objects are always properly initialized when they are created and destroyed when they are no longer needed. A C++ class can provide a member function, called a constructor, that initializes a class object and a member function, called a destructor, that destroys an object. The compiler automatically invokes a constructor whenever a new class object is created and a destructor whenever a class object is destroyed.

### 3.2.1 Encapsulation

Classes are a unit of encapsulation; a class declaration defines a new scope. The type system ensures that member functions of a class can only be applied to instances of a class, thus encapsulating a data structure and the functions that manipulate that structure into a single entity.

The interface to a class is defined by designating members of a class as being either `public` or `private`. A `public` class member can be used without restriction by any program that has a reference to an instance of a class: public data members can be read or written and public member functions may be called. However, a private member can only be accessed from within the class object itself. Private data members can only be accessed by a class member function and private member functions can only be called from within another member function of the class.

### 3.2.2 Constant and Volatile Objects

C++, as well as ANSI C, allows an object to be declared constant or volatile. A constant object (declared `const`) is one that cannot be changed, a volatile object (declared `volatile`) is one whose value can change without being explicitly written to. Constant and volatile can be combined to indicate a

location that cannot be written to, yet whose value can change, for example, a nonsettable, memory mapped clock.

A class object can be declared to be constant or volatile. For example, the declaration:

```
const C x;
```

creates a constant object of class `C` named `x`. Each data member of `x` is a constant object. The semantics of operations on a `const` object are defined by member functions that are explicitly designated as being `const`; only these member functions are allowed to be called in a constant object. Constant member functions allow a programmer to specify the elements of an object's interface that manipulate that object in a manner consistent with a constant declaration. The mechanism used for creating a volatile class object is much like that used for a constant object.

## 3.3 Overloading

A user-defined C++ class, can be used in any situation a language-defined type can be used. This is facilitated by allowing almost all C++ operators to be overloaded so that they apply to user defined types as well as system defined types. Arithmetic operators, comparison operators and data conversion operators (i.e. type casts) are among those operators that can be extended to apply to user-defined data types. Thus one can define the `+` operator to apply to a user defined vector type, string type or matrix type, and to define conversions between each of these types. Conversion between types can be specified in the program via an explicit type cast or automatically applied by the compiler.

## 3.4 Object Oriented Programming

C++ supports object oriented programming via two mechanisms: inheritance and virtual functions. Inheritance is used to support *is-a* relationships between objects. That is relationships such as: *a rat is a rodent*, can be expressed by defining a `rat` class (called a derived class) that inherits a `rodent` class (called a base class). A `rat` object will inherit all behavior specific to `rodent`.

Virtual functions provide for a degree of polymorphism in which the actual behavior of an aspect of an inherited class is defined by the derived class

and not the base class. For example, all rodents get food, but the way a rat gets food is different than the way a guinea pig gets food. This type of relationship is represented via a virtual function.

# 4 The CC++ Language

A sequential C++ program can be converted into a parallel program by using the new constructs provided in CC++. The constructs are:

1. Parallel block.
2. Spawn.
3. Atomic functions.
4. Logical processors.
5. Global pointers.
6. Sync (for synchronizing) variables.

CC++ is a complete notation for writing distributed or parallel programs; there are no constructs such as fair merge, messages, or locks that are outside the notation.

C and C++ programs are valid (albeit sequential) programs in CC++. Therefore, programs in languages (such as Fortran) that can be called from C or C++ can also be called from CC++.

## 4.1 Control Flow

Constructs for sequential execution in CC++ are the same as in C++. CC++ has two constructs for concurrent execution: **parallel blocks**, and **spawn**. A statement in CC++

1. has the syntax of a statement in C++, or,
2. is a parallel block, or,
3. is a spawn statement.

## 4.2 Parallel Blocks

A **parallel block** has the syntax:

`par` *parblock*

where *parblock* has the syntax of a block in C++ with some restrictions given later.

Let `B` be the parallel block: `par { S0 S1 ... Sn }` where `S0 ... Sn` are statements. When execution of `B` is initiated, execution of each of the component statements `S0`, ... , `Sn` of `B` is initiated. Execution of `B` terminates when execution of all its component statements terminate. Therefore, `B` is in execution if and only if at least one of its component statements is in execution.

Thus `B` has the same meaning as the following statement in [OG76]

**cobegin** `S0; S1; ... ; Sn` **coend**

and the following statement in PCN [CT91]:

`{|| S0, S1, ... , Sn }`

The order in which component statements of a parallel block appear within the block is immaterial. For example, `par {S0 S1}` has the same meaning as `par {S1 S0}`.

### Restrictions

1. A parallel block cannot have local variables (with the scope of the parallel block). Therefore, the component statements of a parallel block cannot be declaration statements. (Of course, a component statement `Si` of a parallel block can be a sequential block that has local variables with scope `Si`.)

2. The component statements of a parallel block cannot be labeled.

   There can be no jumps (using `goto`, for instance) from a statement outside a parallel block to a statement within the parallel block. Likewise, there can be no jumps from within a parallel block to statements outside the block. Moreover, there can be no jumps from one component statement `Si` of `B` to a different component statement `Sj` of `B`.

Control can pass to a component statement of `B` in one and only one way: when execution of `B` is initiated, execution of each of the component statements `S0`, ... , `Sn` is initiated. Likewise, control can pass from a component statement of `B` in one and only one way: when all of the component statements of `B` terminate execution, `B` terminates execution. Thus the parallel block is a construct for creating structured parallel programs.

Jumps entirely within one of the component statements `Si` of `B` are permitted if `Si` is a sequential block.

3. A statement in a parallel block must not execute a `return`. Therefore, we place the restriction that a parallel block cannot be a `return block`, where a `return block` is defined (recursively) as a block that contains (i) a `return` statement or (ii) a `return block`.

### 4.2.1 Sequential Block

The syntax of a sequential block in CC++ is the same as the syntax of a block in C++ except that a statement in CC++ can also be a parallel block or a spawn statement. Thus, statements `S0` ... `Sn` are executed in sequence in the sequential block {`S0` ... `Sn`} whereas they are executed concurrently in the parallel block `par` {`S0` ... `Sn`}.

### 4.2.2 Spawn

The syntax of a spawn statement is:

`spawn` *function-call*

where *function-call* has the syntax of a function call in C++.

The execution of a spawn statement creates a new process which executes concurrently with the process executing the spawn. The process created by executing the spawn statement is called the *spawned* process. The process that executes the spawn statement is called the *spawning* process.

Execution of the spawning process continues with the statement (if any) that follows the spawn statement. The spawned process executes the function call specified in *function-call* component of the spawn statement. Values returned by the function are discarded, and not used in the computation.

A difference between `par` blocks and spawn statements is that the `par` block terminates when all its component statements terminate, whereas a spawning process can terminate before or after the spawned process terminates.

**Examples** Consider the blocks:

```
{S0; f(x); S1;}
```

and

```
{S0; spawn f(x); S1;}
```

where `S0;` and `S1;` are statements. In the first block, `S0`, `f(x)` and `S1` are executed sequentially. In the second block, `S0` is executed, and after `S0` terminates execution, `spawn f(x)` is executed. The spawn statement creates a process — the spawned process — and execution of the spawning process continues with `S1`. The spawned process executes `f(x)`.

Thus in the second block, `f(x)` and `S1` are executed concurrently whereas in the first block `S1` is executed after `f(x)` terminates execution.

**Restrictions** Care must be taken in designing the spawned process because it can continue execution even after the spawning process terminates execution. For instance, a process must not throw exceptions to be handled by a process that has terminated execution. Likewise, a process must not reference variables that are local to a terminated process.

## 4.3 Atomicity

An interleaving model of computation is used in CC++: a computation is a *sequence* of operations. A computation of a CC++ program is an arbitrary interleaving of computations of its concurrent processes subject to the requirement of atomicity specified next.

A function can be declared to be an **atomic function** by using the keyword `atomic` in the declaration, as in:

```
atomic void f(int x);
```

An execution of an atomic function is an atomic action: between the initiation of a call to an atomic function and its termination, no other operation occurs in a computation.

**Restrictions**

1. The only data that can be referenced by an atomic member function of an object `p` are member data fields of `p`. An atomic member function of `p` cannot reference members of other objects, nor can it reference static (i.e., global) variables.

   The only data that can be referenced by an atomic static function (i.e., a function that is not a member of an object) are static variables. An atomic static function cannot reference members of objects.

2. An atomic function cannot read a sync value. Therefore, an atomic function cannot suspend execution.

3. Atomic functions must terminate execution in a finite number of steps. Of course, this cannot be checked by the CC++ implementation. The programmer must ensure termination.

**Concurrency of Execution of Atomic Functions** A computation is a *sequence* of operations, and if two atomic operations occur at the "same time" we assume that one of the atomic operations (chosen arbitrarily) occurs before the other in the computation. In practice, many atomic functions can execute at the same time; for instance there can be concurrent execution of atomic functions `p.f(x)` and `q.f(y)` where `p` and `q` are different objects. Concurrently executing atomic functions are independent because one function cannot reference variables referenced by the other. Independent operations can be serialized (in arbitrary order) in the computation.

## 4.4 Fairness

A process is either **executable** or **suspended** at a point in a computation. In CC++, an executable process remains executable until it is executed; the execution of some other process cannot make an executable process become suspended. Executable and suspended processes are discussed later.

Program initiation and termination in CC++ are the same as in C++: program execution starts with initiation of its `main` procedure, and the program terminates execution when `main` terminates execution or when an `exit` or `abort` is called.

The fairness rule implemented by C++ is as follows. At all points `t` in the computation, the following holds:

1. The program terminates (within a finite number of steps from `t`), or
2. for every process `r` that is executable at point `t`: the computation of `r` will progress (within a finite number of steps from `t`).

## 4.5 Sync, Const and Mutable Data

**Introduction** Define a **data unit** as the memory occupied by a fundamental type (e.g. `char`, `int`, `short int`, `long int`, `float`, `double`) or a pointer in C++. Each data unit is **mutable**, **const** (for constant) or **sync**. A mutable data unit has an arbitrary initial value, and can be read and modified an arbitrary number of times. A constant data unit has an unchanging value (unless it is cast as a nonconstant). A sync data unit is a single-assignment or **worm** — write once read many times — data unit. A sync data unit can be assigned a value *at most once* in a computation of a correct CC++ program; an assignment to the same sync data unit more than once in a computation is an error.

**Relationship between Sync and Constant** Sync can be thought of as a constant whose initialization can be delayed, in contrast to a `const` which must be initialized at the point of declaration.

Assigning a value to a sync is called initializing it, in analogy with initializing constants. A sync data unit is defined to be **uninitialized** before it is assigned a value, and **initialized** after it has been assigned a value.

**Process Suspension** A process is either executable or suspended. Associated with each suspended process is a single uninitialized sync data unit; when this data unit becomes initialized the process becomes executable. An executable process `p` can become suspended in one and only one way: `p` is suspended if it reads an uninitialized sync data unit; `p` becomes executable again when the data unit becomes initialized .

Once a sync data unit becomes initialized it remains initialized forever thereafter. Therefore, if a suspended process `p` becomes executable, `p` remains executable until execution of `p` is resumed, and `p` next reads another uninitialized sync data unit.

The value of an uninitialized sync data unit is immaterial since a process that reads an uninitialized sync data unit is suspended until the data unit becomes initialized . The value of an initialized sync data unit is unchanging.

**Aggregate Types** We restrict attention to initialization of sync data units — memory occupied by **fundamental** types or pointers in C++. We do not define initialization of memory occupied by **aggregate** sync types such as `arrays` or `classes`. This is because reading and writing of aggregate types are carried out element-by-element. Thus, some elements of an aggregate sync type can be initialized, while others are uninitialized. A *bit* of an integer cannot be initialized while another bit is uninitialized; an integer is a fundamental type and either the entire integer is initialized or the entire integer is uninitialized.

**Constants and Syncs in Concurrent Computing** An advantage of constants in concurrent computing is that copies of a constant can be made without violating multiple-copy consistency requirements, because a constant remains unchanged. Similarly, copies of syncs — constants with delayed initialization and with flags indicating initialization — can be made, with multiple copy consistency handled in a straightforward way. The only possible inconsistency for sync data is that one copy is uninitialized and the other copy is initialized. In this case, a process that reads the uninitialized copy waits for it to become initialized. The implementation need only guarantee that if one copy is initialized then all copies will become initialized.

## 4.6 Allocation and Deallocation of Storage

In C++, memory is allocated when an object is defined, or when `new t` is executed where `t` is a type, or when memory is allocated by the operating system (by calling `malloc` for instance). Allocation of memory in CC++ is identical to that in C++ except that the memory allocated can be sync (in addition to being constant or mutable). The memory allocated to store a sync object (discussed later) consists of sync data units, and memory allocated to store a mutable object consists of mutable data units.

Storage for sync data units are deallocated (by a garbage collector) some finite time after there are no references to them. The garbage collector col-

lects only sync storage; memory management for mutable and constant storage is as in C++.

Programmers can deallocate storage (by using `delete` for example) but they should do so only if they can prove that there are no references to this storage at the point in the computation at which they are deallocated. Local storage of a block is deallocated when execution of the block terminates (as in C++). Therefore, one block should not reference local variables of a terminated block.

## 4.7 Sync Types

A type in CC++ is either a type in C++ or a **sync version** of a type in C++. A variable that is declared to be a C++ type is a mutable variable or a constant. A variable that is declared to be a sync version of a C++ type is a sync variable.

The type modifier `sync` is used to declare a type to be a sync type. The use of the type modifier `sync` is syntactically identical to the use of the type modifier `const` in C++.

A declaration:

```
sync T A[];
```

where `T` is a C++ type declares `A` to be an array each element of which is of type `sync T`. Programmers cannot declare an array to be partially sync and partially mutable, just as they cannot declare an array to be partially constant and partially mutable.

### 4.7.1 Sync Classes and Structures

In C++, some fields of a structure can be constant types and other fields can be mutable types. Likewise, in CC++, some fields of a structure can be sync types and other fields can be mutable types.

In C++, some member functions of a class can be constant functions (that should not modify data members of the class) and other member functions can be ordinary (non-const) functions. Likewise, in CC++, some member functions of a class can be sync functions and other member functions can be ordinary (non-sync) functions.

**Restrictions**

1. Sync unions are not permitted. Also, no part of a union can be sync.

2. Sync bit fields are not permitted. For instance,

```
sync struct flags{
                    unsigned flagA : 1;
                    unsigned flagB : 1;
                  }
```

   is not permitted.

### 4.7.2 Sync Objects

Since `sync` and `const` are type modifiers, we can declare `sync` versions of objects in the same way that we can declare `const` versions of objects. If `CLS` is a class, then

```
const CLS x
```

declares `x` to be a constant version of an object of class `CLS` in C++. Likewise,

```
sync CLS y
```

declares `y` to be a `sync` version of an object of class `CLS` in CC++.

(An object can be declared to be `const sync`, or equivalently, to be `sync const`, which declares the object to be a const object with the restriction that it cannot be cast as a nonconstant object.)

### 4.7.3 Restrictions on Casting Pointers to Sync

1. A pointer to `sync T` cannot be cast as a pointer to a nonsync type.

2. A pointer to `sync T` can be cast as a pointer to `sync D` if and only if casting a pointer to `T` as a pointer to `D` is legal in C++.

3. A pointer to a fundamental sync type (`int`, `char`, `float`, ...) cannot be cast as a pointer to any other type. Likewise, a pointer to a type `T` cannot be cast as a pointer to a fundamental sync type different from `T`.

## 4.8 Logical Processor Objects

**Introduction** Thus far, we have specified constructs for specifying parallel execution: the (`par` block, `parfor` statement and `spawn` statement), and a mechanism by which parallel program components can communicate and synchronize: (`sync` objects). These constructs are sufficient for constructing parallel programs. There are, however, pragmatic issues that are not addressed by these constructs. In particular, we need to provide:

- a means of abstracting processing resources in the programming language,
- a mechanism for separating algorithmic concerns from resource allocation issues,
- a way to express locality,
- a means for describing heterogeneous computation, and
- a mechanism by which existing C++ codes can be properly composed from within a parallel block.

To resolve these issues, CC++ introduces **logical processor objects**. Logical processor objects are very similar to objects in C++: they have member functions and data members. Logical processor objects are different from C++ objects in one fundamental way: a collection of C++ objects can share information through global and static variables where as every instance of a logical processor object is completely self contained and can interact with other objects only through its public interface.

Logical processor objects address our concerns in the following ways:

- By definition, a logical processor object is an abstraction of the physical processing resources available in a computer system.
- Processor objects are built on the foundation of C++ objects. Therefore, abstract data type and object oriented programming techniques facilitate the abstraction and separation of algorithmic and resource allocation concerns.

- Each processor object is mapped indivisibly onto a physical processing resource. Thus by programming with processor objects, a CC++ program has an explicit means of expressing locality.

- Because a each instance of a logical processor object is self contained, existing C++ programs can be composed in parallel without them interacting unintentionally through shared, global variables.

**The Logical Processor Class** A C++ computation consists of a single instance of a single program. We loosely define a program to be a standalone entity that performs a complete task. While a program in CC++ is the same as a program in C++, a CC++ computation can contain multiple instances of multiples programs. Each instance of a program is represented by a logical processor object. As with any C++ object, a logical processor object has a type associated with it. We call this type the logical processor class.

A C++ program consists of a collection of compilation modules (files) that are linked together, generally by a system utility such as a linker. As part of the linking process, a processor object class is assigned to a program. Existing C++ modules can be linked into a processor object without modification. Since the processor object encapsulates all global and static variables within a program, multiple instances of a pure C++ processor object can be created and execute in parallel without interference.

If one wishes to manipulate a processor object from within a program, then a declaration for the processor object must be provided. A processor object class declaration is just like a regular class declaration, except the keyword `global` is used in combination with the `class` keyword to indicate that the declaration is for a processor object. For example:

```
global class worker {
public:
   int do_work(task);
   int status;
}
```

declares a processor object with one member function and one data member. The public members of a processor object correspond to globally defined

symbols in the corresponding program. In our example, `status` must be a global variable and `do_work` must be a globally accessible function defined when the program is linked.

**Accessing Logical Processors** One of the logical processors is designated the **initiator**. Computation of a CC++ program is initiated by initiating the `main` program on the **initiator** logical processor, and with no other function in execution. It is not possible to execute the `main` program on any logical processor other than the `initiator`.

Additional processor objects can only be created through the use of the C++ dynamic allocation operator: `new`. Initialization of a processor object requires knowing the location of the program text from which the processor object is to be constructed. This information is provided via the C++ constructor argument mechanism. Thus the statement:

```
worker * wPtr = new worker ("~carl/worker");
```

creates a new processor object of class worker and initializes it with the executable found in a file named `worker`. A pointer to the newly created processor object is returned by `new` and placed into the variable `wPtr`. In addition, the global variable `::this` always points to the processor object from which a reference the `::this` is evaluated.

As with any other CC++ object, members of a processor object can be accessed though a pointer to that object. Upon executing the variable declaration shown above, the statements:

```
wPtr->status++;
wPtr->do_work(task(23));
```

will increment the value of status in the newly created `worker` processor object and call the `do_work` function with an argument that is an object of type `task` initialized to the value 23. Note that the `->` operator can be overloaded, eliminating the need for an application program to ever refer directly to a processor object. This provides a flexible, user-defined scheme for addressing logical processors (for examples, see [Fos92]). Examples of how a user might view logical processor ids include an integer, a pair of integers (if the logical processors form a two-dimensional mesh), or an enumerated type.

**Logical Processors and Physical Processors** In addition to providing an argument to the initialization function (i.e. constructor) of an object, C++ allows an argument to be passed to the `new` operator itself. This *placement* argument is used by the CC++ runtime system to specify the low level mapping of logical resources (processor objects) to physical resources (processors). When combined with operator overloading and objects, the placement option to the `new` operator provides the framework from which many mapping strategies can be constructed. For example the statement:

```
worker * wPtr = new (NextNode) worker ("~carl/worker");
```

will create a new processor object of type `worker` and place that processor object on the physical node indicated by the object `NextNode`. The set of physical resource names that are available to a program are implementation specific.

**Linkage** The logical processor class is specified by the collection of linked files that constitute a logical processor object. *All linkage is within a single logical processor object.* There is no name space outside logical processor objects. In C++, one can always refer to a global object named `x` with the expression

```
::x
```

However, in CC++, this expression is interpreted as meaning

```
::this->x
```

that is the value of `x` in the logical processor executing the expression.

A name that has external linkage in the collection of linked files is a *public* member of the processor object. A public member of one object can be referred to in other objects by using the `->` operator. For example if `int x` has external linkage, then `int x` is a public data field of the processor class; a logical processor object can refer to member `x` of another logical processor object `jPtr` as `jPtr->x`.

A name that has internal linkage in the collection of linked files is not a public name of the processor object. For example, if `static int y` appears in a file `f`, then its linkage is internal to `f`, and a logical processor object cannot reference `y` local to file `f` in another logical processor object.

## 4.9 Global Pointers

In C++, a reference to an object can be stored in a pointer variable or a reference variable. In CC++, pointers and references can be used as well. However, in CC++ we must distinguish between the situations in which a pointer or reference variable resides in the same logical processor as the object being referenced or a different logical processor from the object being referenced. The first case is referred to as a *local* pointer or reference, the second case is referred to as a *global* pointer or reference. Thus in CC++, a pointer is a local pointer, a global pointer or a `sync` pointer. For clarity, the following discussion will be limited to pointers, however, it applies to references as well.

A pointer that is not declared to be a global pointer or a sync pointer is a local pointer. In C++, they type of a pointer can be modified with the keyword `const` or `volatile`, to indicate a pointer whose value cannot be changed or one whose value might change spontaneously. In CC++, the type modifiers for a pointer include the keyword **global**. Thus, the statement:

```
int * global g_ptr;
```

declares `g_ptr` to be a global pointer to an integer. Likewise, a `sync` pointer is declared using the keyword **sync** as in:

```
int * sync s_ptr;
```

which declares `s_ptr` to be a sync pointer to an integer. More than one modifier can be used in a declaration, thus the statement:

```
int * global const g_ptr;
```

declares a variable that is a constant, global pointer (i.e. an inter-logical processor pointer whose value cannot be changed).

Global and sync pointers contain two pieces of information: a logical processor id, and an address within that logical processor . A local pointer is the address within a logical processor object, and has no information about the id of the logical processor object itself.

A pointer in one logical processor object that points to a location in another logical processor object must be a global pointer or a sync pointer. A pointer in one logical processor object that points to a location within the

same logical processor object can be any pointer: a local pointer, a global pointer, or a sync pointer.

A variable of type global pointer or sync pointer cannot be assigned to a variable of type local pointer. A variable of type local pointer can be assigned to a variable of type global pointer or sync pointer.

A global pointer cannot be cast as a local pointer.

# 5 Programming examples in CC++

This section presents a few examples that demonstrate the relationship between CC++ and approaches to parallel programming such as concurrent logic programming, functional programming, dataflow languages, monitors, and message-passing. The first three examples show how sequential programs in C can be transformed into C++ programs that execute in parallel.

## 5.1 Transforming Sequential Programs

Consider the sequential C function in Figure 1. The body of this function is a block consisting of a sequence of assignments. Each variable, `a`, `b`, `*x`, `*y` and `*z`, is assigned a value at most once, and a variable does not appear on the right-hand side of an assignment until it has been assigned a value.

```
void f0(int w,  *x,  *y,  *z)
{
  int a,b;
  a = w + 1;
  b = w * 2;
  *x = a-b;
  *y = a*b;
  *z = (*x) + (*y);
}
```

Figure 1: A sequential C++ program.

Figure 2 shows another sequential version of the function `f0`. In this

function, `f1`, the variables that are modified in the function body have been made into `sync` variables.

```
void f1(int w,  sync int *x,  *y,  *z)
{
  sync int a,b;
  {
    a = w + 1;
    b = w * 2;
    *x = a-b;
    *y = a*b;
    *z = (*x) + (*y);
  }
}
```

Figure 2: CC++ sequential program with `sync` variables

Function `f2()` in Figure 3 is identical to `f1()` except that

1. variables that are both read and modified are declared to be `sync` variables, and
2. the sequential block is replaced by a `par` block.

The transformed parallel program is equivalent to the sequential program.

## 5.2 A Discrete-Event Simulation

We begin with a C++ program to simulate a tandem queueing network as shown in Figure 4.

The program:

```
generate_arrivals(int n, float mean, randm rand, cell* p)
```

takes inputs `n`, `mean` and `rand` and makes `p->next` a list of `n` arrival times with the mean interarrival time equal to `mean` and using a random number generator specified by object `rand`.

```
void g(int w, sync int *sync x,  *sync y,  *sync z)
{
  sync int a, b;
  par { a = w + 1;
b = w * 2;
*x = a-b;
*y = a*b;
*z = (*x) + (*y);
      }
}
```

Figure 3: A parallel CC++ version of the sequential function `f0`

The program

```
generate_service(int n, float mean, randm rand, cell* p)
```

is similar to `generate_arrivals` except that it sets `p->next` to be a list of service times.

The program `void simulate_queue(cell* a, cell* s, cell* d)` simulates a first-come-first-served queue with arrival times specified by list `*a`, service times specified by list `*s`, and it computes the departure times from the queue and stores them in list `*d`. The program to simulate a queue uses the following equation:

$$departure_i = service_i + max(departure_{i-1}, arrival_i)$$

where $departure_i$, $service_i$, and $arrival_i$ are the departure time, service time, and arrival time of the $i$-th job in the queue.

The program is given below.

The main program simulates a sequence of queues, with departures from one queue becoming arrivals to the next. The variables `d0`, and `d1` represent departures from queues 0 and 1, respectively, and variables `s0` and `s1` represent service times at queues 0 and 1, respectively.

The main program is given below:

```
void generate_arrivals(int n, float mean, randm rand, cell * p)
{
  cell * q = p;
  float t = 0.0;
  int i;

   for (i=0; i<n; i++) {
     t = t+rand.generate(mean);
     q->next = new cell(t);
     q = q->next;
   }
  q->next = (cell*) NULL;
}
```

Figure 4: Sequential function for arrival time generation

**From Sequential to Parallel** First, consider the sequential program. The initial values of lists `a`, `s0`, `d0`, `s1`, `d1`, are arbitrary, and elements of the lists are assigned values once. Therefore, we can safely convert these variables into `sync` variables. For example, we can change the declaration of `a` from

```
cell * a;
```

to

```
cell * sync a;
```

Pointers to `sync` must be `global` pointers, therefore we change `q` in `generate_arrivals` and `pa`, `ps`, `pd` in `simulate_queue` to global. For example, we can change the declaration of `q` from

```
cell * q;
```

to

```
cell * global q;
```

Finally change the sequential block in `main` to a parallel block:

```
void simulate_queue(cell* a, cell* s, cell* d)
{
  float t = 0.0;

  cell* pd = d;
  cell* pa = a->next;
  cell* ps = s->next;

  while( (pa != (cell*) NULL) && (ps != (cell*) NULL)) {
    t = (ps->value) + max(t,pa->value);
    pd->next = new cell(t);

    pa = pa->next;
    ps = ps->next;
    pd = pd->next;
  };

  pd->next = (cell*)NULL;
}
```

Figure 5: Sequential function for computing simulation times

```
main()
{
  cell *a, *s0, *d0, *s1, *d1;

  a = new cell(0.0);
  generate_arrivals(n, mean_interarrival, rand, a);

  s0 = new cell(0.0);
  generate_service(n, mean_service0, rand, s0);

  d0 = new cell(0.0);
  simulate_queue(a,s0,d0);

  s1 = new cell(0.0);
  generate_service(n, mean_service1, rand, s1);

  d1 = new cell(0.0);
  simulate_queue(d0,s0,d1);

}
```

Figure 6: Main program for sequential simulation

## 5.3 How A Process Can Halt Another

Consider the following modification of `generate_arrivals`. In the program given earlier, $n$ arrivals are generated where $n$ is an input variable. Now, we want to modify the program to continue generating arrivals until some process initializes a `sync` variable `done` to `TRUE`.

We introduce a boolean (mutable) variable `over` which is initially `FALSE` and becomes `TRUE` after `done` is initialized.

```
main()
{
  int over = FALSE;
  sync int done;
```

```
main()
{
  cell *sync a, *sync s0, *sync d0, *sync s1, *sync d1;
  // begin parallel block
  par {
    a = new cell(0.0);
    generate_arrivals(n, mean_interarrival, rand, a);

    s0 = new cell(0.0);
    generate_service(n, mean_service0, rand, s0);

    d0 = new cell(0.0);
    simulate_queue(a,s0,d0);

    s1 = new cell(0.0);
    generate_service(n, mean_service1, rand, s1);

  } // end parallel block
}
```

```
  par {
    over = done;
    generate_arrivals(..,over,..);
    process_that_sets_done( $\ldots$, done, $\ldots$);
    $\ldots$
    }
  $\ldots$
}
```

Replace the `for` loop in `generate_arrivals` by the `while` loop:

```
while(!over){ ...  }
```

## 5.4 Parafunctional Programming

The next program is a divide-and-conquer algorithm that produces a list of cells. An example of such a program is one that produces a list of cells whose data fields are the sequence of integers from `low` to `high`.

```
sync cell * sync f(int low, high)
{
  int middle = (high-low)/2;
  if (high==low)
    return g(middle);
  else if (high-low <= threshold)
    return merge(f(low,middle), f(middle+1,high));
  else {
    sync cell *sync result = (sync cell *sync) new sync cell;
    spawn *result = *merge(f(low,middle),f(middle+1,high));
    return result;
  }
}
```

Consider the evaluation of a statement:

```
x = h(f(0,10),f(30,50));
```

where `h` is a function that has two arguments both of which are of type `sync cell *sync`. The evaluation of `f(0,10)` will return a pointer, say `p` to a `sync cell`, and have the side effect of spawning a statement that will create the list `*p`. Likewise, the evaluation of `f(30,50)` will return a pointer, say `q` to a `sync cell`, and have the side effect of spawning a statement that will create the list `*q`. Thus, the evaluation of the arguments of `h` have the side effect of spawning statements that "fill in" the values these arguments.

Think of program `f` in a parallel function library. We can write a sequential program, that calls functions from this library. The user of the library doesn't have to be concerned with sequential programming at all, but nevertheless function arguments will be evaluated in parallel.

## 5.5 Distributed Abstract Data Types

Consider:

```
array class {
public:
  increment();
private:
  int size;
  int a[size];
}

void array::increment()
{
  for( int i=0; i < size; i++)
    a[i]++;
}
```

and another class:

```
array distributed_class {
public:
  void increment();
private:
  int size;
  int number_of_blocks;
  int size_of_block;
  int *a[number_of_blocks];
}

void array::increment()
{
  parfor(int i=0; i < number_of_blocks; i++)
    for(int j=0; j < size_of_block; j++)
      a[i][j]++;
}
```

Consider the following program stub:

```
array x;
distributed_array y;
x.increment();
y.increment();
```

The function calls `x.increment()` and `y.increment()` have the same effect, though the former call is implemented by a sequential algorithm, and the latter by a distributed algorithm.

Here is another example, where the user of a library can write a sequential program, and have the program execute in parallel.

## 5.6 Monitors

Let us implement a message-passing channel with asynchronous sends and asynchronous receives. This example demonstrates the use of atomicity.

```
struct msg_cell{
  msg *data;
  msg_cell *next; }
```

We will create a list of `msg_cell` where the `data` field of each cell will point to a message.

```
struct ptr_cell{
  msg *sync *global data;
  ptr_cell *next; }
```

We use a list of `ptr_cell` where the data field of each cell points to a sync pointer to a message.

```
class channel{
 private:
   ptr_cell *recv_front, *recv_back;
   msg_cell *sent_front, *sent_back;

 public:
   atomic void send(msg m);
   atomic void receive(msg *sync *p);
}
```

Here `recv_front` and `recv_back` point to the front and back of the receive queue which is a queue of `ptr_cell`. Likewise `sent_front` and `sent_back` point to the front and back of the sent queue which is a queue of `msg_cell`.

Both sending and receiving are asynchronous. The sender does not block if there is no pending receive. Likewise, the receiver does not block if there is no message for it. The receiver executes `receive(p)` where `p` is of type `msg *sync *`; if `*p` is uninitialized then there is no message corresponding to this receive; if `*p` is initialized then `*p` is the address of the message that is received.

An empty queue is represented by a single *sentinel* cell that has an arbitrary data field. Thus, in an empty queue, `recv_front = recv_back` and both point to the sentinel cell. The last cell in the queue points to `NULL`; thus, `recv_back->next = NULL`.

An invariant of the program is that the receive queue or the sent queue is empty. Initially, both queues are empty. The sent queue contains the sequence of messages that have been sent and not yet received. The receive queue contains the sequence of pointers that will be assigned to messages when they arrive.

The receive operation is the analog of the send operation, interchanging the use of receive queues and sent queues.

# 6 Implementing CC++

An initial implementation of CC++ has recently been completed. In this section, we will discuss the significant aspects of the implementation. The goal of our implementation is functionality, not performance; there are clearly many avenues for optimization that can and will be pursued.

Since the relationship between CC++ and C++ is so close, any implementation of CC++ should be based on an existing C++ implementation. Our strategy is to preprocess a CC++ program into a proper C++ program. The resulting C++ program is then translated into an executable form by an unmodified C++ compiler. The components of the implementation that are specific to CC++ consist of: 1) a CC++ to C++ translator, 2) a set of C++ class definitions that are used by the C++ compiler and 3) a CC++ runtime library.

The primary mechanisms that the CC++ implementation must provide

```
atomic void channel::send(msg m)
{
  msg *q = new msg(m);
  // q points to an area of memory containing a copy of m.

  // if the receive queue is empty, append m to the sent queue
  if(recv_front->next = (ptr_cell *) NULL) {
    sent_back->next = new msg_cell(q,(msg_cell*) NULL);
    // sent_back ->next points to a msg_cell whose data field
    // points to msg m, and whose next field is NULL.
    sent_back = sent_back->next;
  } else {
    // Since the receive queue is not empty, there is at least one
    // pending receive; so make the first pending receive pointer
    // point to m.
    ptr_cell *r = recv_front->next;
   // r points to first pending receive.
    recv_front->next = r->next;
   // r is no longer in the receive queue.
    *(r->data) = (msg *sync) q;
    delete r;
  }
}
```

are:

- multiple threads of control (for `par` blocks and `spawn` statements,
- global references, and
- single assignment semantics for all fundamental data types (integer, character, floats, etc.) and pointers.

Multiple threads of control are provided by any lightweight process library; an existing C++ thread package such as [BLL88, SS87] could also be used. The minimal capability required by the thread library is the ability to

```
atomic void channel::receive(msg *sync *p)
{
  // if the sent queue is empty, append p to the receive queue
  if(sent_front->next = (msg_cell *) NULL) {
    recv_back->next = new ptr_cell(p,(ptr_cell*)NULL);
    // recv_back-¿next points to a ptr_cell whose data field
    // is p, and whose next field is NULL.
    recv_back = recv_back->next;
  } else {
    //  Since the sent queue is not empty, there is at least one
    // unreceived message; so make p point to the first unreceived
    // message.
    msg\_cell *r = sent\_front->next;
    // r points to cell pointing to first unreceived message.
    sent\_front->next = r->next;
    // r is no longer in the sent queue.
    *p = (msg *sync) r->data;
    delete r;
  }
}
```

create new threads, uniquely identify a thread via an identifier and suspend and resume thread execution via its identifier. Certain execution environments, such as shared memory multiprocessors, also require the ability to perform an atomic action.

The implementation of CC++ global references varies greatly depending on the target architecture. Some parallel computers have hardware support for a single shared address space. On such machines, the CC++ implementation maps global references into regular references. However, many scalable parallel architectures do not directly support shared address spaces. While operating systems can support a global address space with page level sharing [LS89], finer resolutions are required for CC++.

On machines that don't provide a global address space, global pointers must be implemented in software. A global pointer is represented by an identifier for the address space being pointed into and the address within

that address space. Operator overloading is used to convert a request for the contents of a global pointer into a message requesting its value from the appropriate processor. If the global pointer is to a sync object, the entire object can be cached on the processor on which the global pointer located. Obviously, using a global pointer to reference an object between logical processor objects will cost more that a local reference. However, as discussed in Section 2.1, locality of reference cannot be ignored.

Like global references, the implementation of sync objects will depend on the underlying hardware support. Some parallel computer architectures, such as the Dally' J-Machine and Smith's Tera, have hardware support for determining if a memory location has been written. However, on most computers, sync objects must be implemented in software. The CC++ implementation does this by associating a tag with every sync data element.

### 6.1 Converting CC++ to C++

To convert CC++ into C++, the CC++ program is parsed, constructing a parse tree. The parse tree is then modified, converting it into a C++ parse tree and then written out in textual form. The CC++ parser is a modified version of the GNU C++ (G++) [Tie91] parser. There are some minor syntactic limitations on the C++ accepted by G++ and the current CC++ implementation inherits these.

The hardest part of the transformation process is to ensure that the semantic restrictions outlined in Section 4 are checked as carefully as possible. The actual transformations required are straight forward.

## 7 Related Research

In our work, we have focused on language mechanisms that facilitate the design of interfaces for parallel composition. The design of CC++ draws ideas from a wide range of parallel programming languages. These include data flow languages with single-assignment variables [TE68, Ack82], remote procedure calls [TA90], message passing [Sei91], actors [Agh86], concurrent logic programming [FT90, Ued86, Sha86] and compositional languages, particularly PCN [CT91]. In this section, we discuss how significant aspects of CC++ relate to other parallel programming notations.

Parallel composition as defined by `par` blocks can be found in a number of other parallel programming notations. For example, a `par` is equivalent to the use of *cobegin* and *coend* in [OG76] and the parallel composition operator in PCN [CT91].

The use of `par` blocks differs from most other concurrent object-oriented languages in that with a `par` block, multiple threads of control exist within a single object. Other languages tend to associate thread creation with object creation [BLL88, Gri91, SS87, Agh86]. Consequently, only one thread of control is ever associated with an object. The `spawn` statement in CC++ can be used to achieve the same effect.

The advantage of `par` blocks and `parfor` statements over the tying thread creation to object creation approach is twofold. First, these statements are block-oriented, and they make parallelism within a block explicit; there is no question as to which statements execute in parallel and which in sequence. The second advantage is that one can associate a post condition with a `par` block or a `parfor` statement. When the statement terminates, the post condition can be asserted. This simplifies the process of reasoning about the behavior of the program.

Reasoning about the behavior of a program containing a `spawn` statement is more difficult. Because there is no way of knowing when the thread started by the `spawn` starts or completes, assertions about the `spawn` statement must state that a condition will hold at some unknown point in the future. Thus one has no choice but to resort to a temporal operator such as the *leads-to* operator.

No doubt, some readers will have recognized sync variables as being single assignment variables from dataflow languages [M+85] or from languages based in concurrent logic programming [FT89]. A reference to a structure which contains `sync` objects behaves much like an I-Structure in the dataflow language ID [AT80]. `sync` variables differ in that the `sync` attribute can be extended to abstract and concrete data types as definable in C++. In addition, `sync` variables differ from variables in programming languages such as Strand [FT89] and PCN [CT91] in that the blocking rule for `sync` variables prohibits the use of variable-to-variable assignment found in these languages.

Many concurrent object-oriented languages [Agh86, AS88, WHD89] use function call as the basis for communication. In actor based languages, a function call is interpreted as sending a message to the target object to perform the requested operation. The arguments in the function call are

passed to the target object as well. The function call terminates immediately, without a waiting for a response from the target object. Applying this approach to C++ is problematic in that this approach changes the meaning of function call. By associating communication with shared variables and assignment, the semantics of all of the underlying operations in C++ are preserved. Finally, we note that actor type semantics of function call can be achieved either through the use of the `spawn` statement within the body of a member function or assignment to a `sync` variable whose value is read by a nondeterministic fair merger.

It is important to recognize that a `sync` variable in CC++ is a pure single assignment variable and not a logical variable as found in concurrent logic programming languages such as Strand [FT90], FCP [Sha86], GHC [Ued86] or Parlog [Gre87] or compositional languages such as PCN [CT91]. In particular, the assignment `x = y` suspends until y has a value; variable-to-variable assignments are not made. Consequently, structured `sync` data behaves more like an I-Structure [AT80] from the dataflow language Id [Ack82] than a tuple from a logic programming language. The use of single assignment variables in place of logical variables has the advantages that assignment semantics are completely consistent with C++, and that pointer dereferencing is not required prior to variable use. The disadvantage is that some concurrent logic programming techniques, such as the short circuit technique [Tak89] become sequentialized. This is not a significant drawback, however, because termination of parallel blocks is easily determined.

CC++ has many ideas in common with the parallel programming language PCN [CT91] which in turn draws heavily from committed choice concurrent logic programming languages such as Strand [FT90]. There are, however, fundamental differences between them. These include:

- CC++ provides a general shared-memory programming model. This includes having pointers to data objects anywhere in distributed memory.
- CC++ is a tiny extension to an object-oriented language whereas PCN is a new language. Indeed, most of the power of CC++ derives from C++.
- Remote procedure call is a primitive operation in CC++.

- There are no nondeterministic language constructs in CC++ as opposed to PCN. Nondeterminism in CC++ can be obtained only through interleaving of atomic actions.
- PCN permits `x = y` as an equality. CC++ treats all assignment operators as assignment of value.

# 8 Summary

The six additions to C++ allow for the systematic development of concurrent programs. The central thesis of this effort is that it is possible to derive correct efficient parallel programs and support a variety of parallel programming paradigms in C++, using C++ tools such as programming environments and libraries, and six new constructs that are similar to existing constructs in C++.

# 9 Acknowledgment

The implementation of CC++ was done with the assistance of Mei Su, Tal Lancaster, Pete Carlin, Marc Pomerantz, Julia George and Ranjit Mathews. Thanks to Ian Foster and Craig Lee for their suggestions and for reviewing various versions of this document.

The research on C++ object libraries for concurrent computation is funded by DARPA under grant N00014-91-J-4014. The research on compositional concurrent notations is funded by the NSF Center for Research on Parallel Computing under grant CCR-8809615.

# References


[Ack82] William B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, feb 1982.

[Agh86] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[AS88] William C. Athas and C.L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, aug 1988.

[AT80] Arvind and R.E. Thomas. I-Structures: An efficient data structure for functional languages. Technical Report TM-178, MIT, 1980.

[BLL88] Brian Bershad, Edward Lazowska, and Henry Levy. Presto: A system of object-oriented parallel programming. *Software: Practice and Experience*, 18(8):713–732, aug 1988.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design*. Addison-Wesley, 1988.

[CT91] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming*. Bartlett and Jones, 1991.

[Fos91] Ian Foster. Program transformation notation: A tutorial. Technical Report ANL-91/38, Argonne National Laboratory, 1991.

[Fos92] Ian Foster. Information hiding in parallel programs. Technical Report MCS-P290-0292, Argonne National Laboratory, 1992.

[FT89] Ian Foster and Stephen Taylor. *STRAND: New Concepts in Parallel Programming*. Prentice Hall, 1989.

[FT90] Ian Foster and Stephen Taylor. *Strand: New Concepts in Parallel Programming*. Prentice Hall, 1990.

[FT92] Ian Foster and Steve Tuecke. Parallel programming with PCN. Technical Report ANL-91/32, Argonne National Laboratory, 1992.

[Gre87] Steve Gregory. *Parallel Logic Programming in PARLOG*. International Series in Logic Programming. Addison-Wesley, 1987.

[Gri91] Andrew S. Grimshaw. An introduction to parallel object-oriented programming with Mentat. Computer Science Report TR-91-07, University of Virginia, 1991.

[Hoa74] C.A.R Hoare. Monitors: An operating system structuring concept. *cacm*, 17(10):549–557, oct 1974.

[Hou92] C. Houck. Run-time system support for distributed actor programs. Master's thesis, University of Illinois at Urbana-Champaign, 1992.

[LS89] Kai Li and Richard Schaefer. A hypercube shared virtual memory system. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 125–132, aug 1989.

[M+85] J. McGraw et al. SISAL: Streams and iteration in a single assignment language, language reference manual, version 1.2. Technical Report M-146, LLNL, mar 1985.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall International, 1988.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(1):319–340, 1976.

[PBH77] Per Brinch Hansen. *The Architecture of Concurrent Programs*. Prentice-Hall, 1977.

[Sei91] Charles Seitz. *Developments in Concurrency and Communication*, chapter 5, pages 131–200. Addison Wesley, 1991.

[Sha86] Ehud Shapiro. Concurrent Prolog: A program report. *IEEE Computer*, 19(8):44–58, aug 1986.

[SS87] Bjarne Stroustrup and Jonathan Shopiro. A set of C++ classes for co-routine style programming. In *Proceedings of the USENIX C++ Workshop*, nov 1987.

[TA90] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *ACM Operating Systems Review*, 24(3), jul 1990.

[Tak89] Akikazu Takeuchi. How to solve it in Concurrent Prolog. Unpublished note., 1989.

[TE68] L. Tesler and H. Enea. A language for concurrent processes. In *Proceedings of AFIPS SJCC*, number ANL-91/38, 1968.

[Tie91] Michael D. Tiemann. *User's Guide to GNU C++*. Free Software Foundation, Inc., oct 1991.

[Ued86] Kazunori Ueda. Guarded horn clauses. In *Logic Programming '85*, pages 168–179. Springer-Verlag, 1986.

[WHD89] Andrew A. Chien Waldemar Horwat and William J. Dally. Experience with CST: Programming and implementation. In *SIGPLAN 89 Conference on Programming Language Design and Implementation*, 1989.