Draft Specification of Transactional Language Constructs for C++

Version: 1.1

February 3, 2012

Transactional Memory Specification Drafting Group

Editors: Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, and Justin Gottschlich, Intel

Important Legal Terms

We welcome feedback on the contents of this documentation and on the specification in particular. To preserve our ability to incorporate such input into future versions of the specification and to standardize future versions, we provide this specification under license terms that require any party who provides substantive feedback to grant the authors a license. The feedback license terms apply only if you choose to provide feedback.

License to feedback: You may, at your sole discretion and option, communicate written suggestions or other ideas about potential changes to the specification, either directly to any author or in a forum accessible to the authors, such as a blog or mailing list. Any such communications will be deemed feedback. By the act of providing feedback you indicate your agreement that you grant to each of the authors, under your copyright and patent rights in such feedback, a worldwide, non-exclusive, royalty-free, perpetual, and sublicensable (with the authority to authorize the granting of sublicenses) license to (a) modify and create derivative works of such feedback, (b) copy, distribute, perform and display such feedback and derivative works thereof, and (c) use, make (including design and develop), have made (including have designed and have developed), import, and directly and indirectly sell and offer to sell products incorporating the feedback in whole or in part, provided that the license rights granted in this sub-section (c) shall apply solely to the feedback as originally furnished by you and solely to the extent that such feedback is incorporated (in whole or part) into the specification. You agree that any feedback you provide is not confidential information.

As used in these terms, “you” refers to an individual’s employer when that individual is acting within the scope of their employment. In such a case the individual represents to the authors that they are authorized to bind their employer to these terms.

Contributors

This specification is the result of the contributions of the following people:

Ali-Reza Adl-Tabatabai, Intel
Kit Barton, IBM
Hans Boehm, HP
Calin Cascaval, IBM
Steve Clamage, Oracle
Robert Geva, Intel
Justin Gottschlich, Intel
Richard Henderson, Red Hat
Victor Luchangco, Oracle
Virendra Marathe, Oracle
Maged Michael, IBM
Mark Moir, Sun
Ravi Narayanaswamy, Intel
Clark Nelson, Intel
Yang Ni, Intel
Daniel Nussbaum, Oracle
Torvald Riegel, Red Hat
Tatiana Shpeisman, Intel
Raul Silvera, IBM
Xinmin Tian, Intel
Douglas Walls, Oracle
Adam Welc, Intel
Michael Wong, IBM
Peng Wu, IBM

Feedback

We welcome feedback on this specification. The feedback should be directed to the TM & Languages discussion group – `http://groups.google.com/group/tm-languages`.

Contents

1. Overview

This specification introduces transactional language constructs for C++, which are intended to make concurrent programming easier by allowing programmers to express compound statements that do not interact with other threads, without specifying the synchronization that is required to achieve this. We briefly describe the features introduced in this specification below.

This specification builds on the C++11 specification. As such, the constructs described in this specification have well-defined behavior only for programs with no data races. This specification specifies how the transactional constructs contribute to determining whether a program has a data race (Section 2.1).

The `__transaction_relaxed` keyword (Section 3) can be used to indicate that a compound statement should execute as a relaxed transaction; that is, the compound statement does not observe changes made by other transactions during its execution, and other transactions do not observe its partial results before it completes. Relaxed transactions may contain arbitrary non-transactional code and thus provide interoperability with existing forms of synchronization. Relaxed transactions, however, may appear to interleave with non-transactional actions of other threads.

To enforce a more strict degree of transaction isolation, we introduce atomic transactions represented by the `__transaction_atomic` keyword (Section 4). An atomic transaction executes a single indivisible statement; that is, it does not observe changes made by other threads during its execution, and other threads do not observe its partial results before it completes. Furthermore, the atomic transaction statement takes effect in its entirety if it takes effect at all.

Two additional syntactic features allow the programmer to specify expressions (Section 5) and functions (Section 6) that should execute as relaxed or atomic transactions.

To make the atomic transaction behavior possible, the compiler enforces a restriction that an atomic transaction must contain only "safe" statements (Section 4.2), and functions called within atomic transactions must contain only safe statements; such functions – and pointers to such functions – must generally be declared with the `transaction_safe` attribute. Under certain circumstances, however, functions can be inferred to be safe, even if not annotated as such (Section 3.2). This is particularly useful for allowing the use of template functions in atomic transactions. Functions may be annotated with the `transaction_unsafe` attribute to prevent them from being inferred as `transaction_safe`. This is useful to prevent a function from being used in an atomic transaction if it is expected that the function may not always be safe in the future. The attributes on a virtual function must be compatible with the attributes of any base class virtual function that it overrides (Section 10). To minimize the burden of specifying function attributes on member functions, class definitions can be annotated with default attributes for all member functions, and these defaults can be overridden (Section 11).

An atomic transaction statement can be cancelled using the `__transaction_cancel` statement (Section 8), so that it has no effect. Cancellation avoids the need to write cleanup code to undo the partial effects of an atomic transaction statement, for example, on an error or unexpected condition. A programmer can throw an exception from the cancelled transaction statement by combining the cancel statement with a `throw` statement to form a cancel-and-throw statement (Section 9).

Atomic transactions can be nested, but a programmer can prohibit a transaction statement from being nested by marking it as an outermost atomic transaction using the `outer` attribute (Section 4.1). A cancel or a cancel-and-throw statement can be annotated with the `outer` attribute to indicate that the outermost atomic transaction should be cancelled (Sections 8.1 and 9.1). Such cancel and cancel-and-throw statements can execute only within the dynamic extent of a transaction statement with the `outer` attribute. The `transaction_may_cancel_outer` attribute for functions and function pointers facilitates compile-time enforcement of this rule.

If an exception escapes from an atomic transaction statement without it being explicitly cancelled, the atomic transaction takes effect. Programmers can guard against subtle bugs caused by exceptions escaping a transaction statement unexpectedly by using noexcept specifications (Section 7) to specify if exceptions are (or are not) expected to be thrown from within an atomic transaction. A runtime error occurs, which leads to program termination, if an exception escapes the scope of an atomic transaction that has a noexcept specification specifying no exceptions may escape its scope.

Appendix A includes a grammar for the new features. Appendix B discusses dependencies between features, to assist implementers who might be considering implementing subsets of the features described in this document or enabling features in different orders. Appendix C discusses several possible extensions to the features presented in this specification. Appendix D describes changes compared to the previous version of the specification.

2. Transaction statement

The `__transaction_relaxed` or the `__transaction_atomic` keyword followed by a compound statement defines a *transaction statement*, that is, a statement that executes as a transaction:

`__transaction_relaxed` *compound-statement*
`__transaction_atomic` *compound-statement*

In a data-race-free program (Section 2.1), all transactions appear to execute sequentially in some total order. This means that transactions execute in isolation from other transactions; that is, the individual operations of a transaction appear not to interleave with individual operations of another transaction.

[Note: *Although transactions behave as if they execute in some serial order, an implementation (i.e., compiler, runtime, and hardware) is free to execute transactions concurrently while providing the illusion of serial ordering.*]

A transaction statement defined by the `__transaction_relaxed` keyword specifies a *relaxed transaction* (Section 3). A transaction statement defined by the `__transaction_atomic` keyword specifies an *atomic transaction* (Section 4). Relaxed transactions have no restrictions on the kind of operations they may contain, but provide only basic isolation guarantee of all transactions – they appear to execute sequentially with respect to other transactions (both relaxed and atomic). Relaxed transactions may appear to interleave with non-transactional operations of another thread. Atomic transactions provide a stronger isolation guarantee; that is, they do not appear to interleave with any operations of other threads. Atomic transactions, however, may contain only “safe” code (Section 4.2).

A `goto` or `switch` statement must not be used to transfer control into a transaction statement. A `goto`, `break`, `return`, or `continue` statement may be used to transfer control out of a transaction statement. When this happens, each variable declared in the transaction statement will be destroyed in the context that directly contains its declaration.

The body of a transaction statement may throw an exception that is not handled inside its body and thus propagates out of the transaction statement (Section 7).

2.1 Memory model

Transactions impose ordering constraints on the execution of the program. In this regard, they act as synchronization operations similar to the synchronization mechanisms defined in the C++11 standard (i.e., locks and C++11 atomic variables). The C++11 standard defines the rules that determine what values can be seen by the reads in a multi-threaded program. Transactions affect these rules by introducing additional ordering constraints between operations of different threads.

[Brief overview of C++11 memory model:

An execution of a program consists of the execution of all of its threads. The operations of each thread are ordered by the *"sequenced before"* relationship that is consistent with each thread's single-threaded semantics. The C++11 library defines a number of operations that are specifically identified as *synchronization operations.* Synchronization operations include operations on locks and certain atomic operations (that is, operations on C++11 atomic variables). In addition, there are `memory_order_relaxed` atomic operations that are not synchronization operations. Certain synchronization operations *synchronize with* other synchronization operations performed by another thread. (For example, a lock release synchronizes with the next lock acquire on the same lock.)

The "sequenced before" and "synchronizes with" relationships contribute to the *"happens before"* relationship. The "happens-before" relationship is defined by the following rules:

1. If an operation A is sequenced before an operation B then A happens before B.
2. If an operation A synchronizes with an operation B then A happens before B.
3. If there exists an operation B such that an operation A happens before B and B happens before an operation C then A happens before C.

(In the presence of `memory_order_consume` atomic operations the definition of the "happens-before" relationship is more complicated. The "happens-before" relationship is no longer transitive. These additional complexities, however, are orthogonal to this specification and are beyond the scope of a brief overview.) The implementation must ensure that no program execution demonstrates a cycle in the "happens before" relation.

Two operations *conflict* if one of them modifies a memory location and the other one accesses or modifies the same memory location. The execution of a program contains a *data race* if it contains two conflicting operations in different threads, at least one of which is not an atomic operation, and neither happens before the other. Any such data race results in undefined behavior. A program is *data-race-free* if none of its executions contains a data race. In a data-race-free program each read from a non-atomic memory location sees the value written by the last write ordered before it by the "happens-before" relationship. It follows that a data-race-free program that uses no atomic operations with memory ordering other than the default `memory_order_seq_cst` behaves according to one of its sequentially consistent executions.]

Outermost transactions (that is, transactions that are not dynamically nested within other transactions) appear to execute sequentially in some total global order that contributes to the "synchronizes with" relationship. Conceptually, every outermost transaction is associated with StartTransaction and EndTransaction operations, which mark the beginning and end of the transaction.[1] A StartTransaction operation is sequenced before all other operations of its transaction. All operations of a transaction are sequenced before its EndTransaction operation. Given a transaction T, any operation that is not part of T and is sequenced before some operation of T is sequenced before T's StartTransaction operation. Given a transaction T, T's EndTransaction operation is sequenced before any operation A that is not part of T and has an operation in T that is sequenced before A.

There exists a total order over all StartTransaction and EndTransaction operations called the *transaction order*, which is consistent with the "sequenced-before" relationship. In this order,

[1] We introduce these operations purely for the purpose of describing how transactions contribute to the "synchronizes with" relationship.

transactions do not interleave; that is, no StartTransaction or EndTransaction operation executed by one thread may occur between a matching pair of StartTransaction and EndTransaction operations executed by another thread.

The transaction order contributes to the “synchronizes with” relationship defined in the C++11 standard. In particular, each EndTransaction operation synchronizes with the next StartTransaction operation in the transaction order executed by a different thread.

[Note: *The definition of the “synchronizes with” relation affects all other parts of the memory model, including the definition of the “happens before” relationship, visibility rules that specify what values can be seen by the reads, and the definition of data race freedom. Consequently, including transactions in the “synchronizes with” relation is the only change to the memory model that is necessary to account for transaction statements. With this extension, the C++11 memory model fully describes the behavior of programs with transaction statements.*]

[Note: *A shared memory access can form a data race even if it is performed in a transaction statement. In the following example, a write by thread T2 forms a data race with both read and write to x by Thread T1 because it is not ordered with the operations of Thread T1 by the “happens-before” relationship. To avoid a data race in this example, a programmer should enclose the write to x in Thread T2 in a transaction statement.*

<table>
<tr><th>Thread T1</th><th>Thread T2</th></tr>
<tr><td><pre>__transaction_relaxed {
      t = x;
      x = t+1;
}</pre></td><td><pre>
x = 1;</pre></td></tr>
</table>

]

[Note: *The C++11 memory model has consequences for compiler optimizations. Sequentially valid source-to-source compiler transformations that transform only code between synchronization operations (which include StartTransaction and EndTransaction operations), and which do not introduce data races, remain valid. Source-to-source compiler transformations that introduce data races (e.g., hoisting load operations outside of a transaction) may be invalid depending on a particular implementation of this specification.*]

3. Relaxed transactions

A transaction statement that uses the `__transaction_relaxed` keyword defines a *relaxed transaction*. We call such a statement a *relaxed transaction statement*:

`__transaction_relaxed` *compound-statement*

A relaxed transaction is a compound statement that executes without observing changes made by other transactions during its execution. Furthermore, other threads’ transactions do not observe partial results of concurrently executing transactions. Programmers can think of a relaxed transaction statement as a sequence of operations that do not interleave with the operations of other transactions, which simplifies reasoning about the interaction of concurrently executing transactions of different threads.

Relaxed transactions have no restrictions on the kind of operations that can be placed inside of them and, thus allow any non-transactional code to be wrapped in a transaction. This makes relaxed transactions flexible with regard to their usability, thereby allowing them to communicate with other threads and the external world (e.g., via locks, C++11 atomic variables, volatile variables or I/O) while still isolating them from other transactions. However, relaxed transactions that contain such external world operations are not guaranteed isolation, even in data-race-free programs. Other threads that communicate with a transaction can observe partial results of the transaction, and the transaction can observe actions of other threads during its execution.

The following example illustrates a data-race-free program in which a relaxed transaction synchronizes with another thread via access to a C++11 atomic variable: Note that accesses to variable `x` in Thread 1 do not form data races with accesses to `x` in Thread 2 because operations on C++11 atomic variables cannot create a data race:

Initially atomic<int> x = 0;	
Thread T1	Thread T2

__transaction_relaxed {
   x = 1;
 	
  while (x != 0) {}
}
	
	while (x != 1) {}
	x = 0;

Relaxed transactions appear to interleave with non-transactional actions of other threads only when they perform non-transactional forms of synchronization, such as operations on locks or C++11 atomic variables. Transactions that do not execute such actions appear to execute atomically, that is, as single indivisible operations.

Relaxed transactions may execute operations with side effects that the system cannot roll back. We refer to such operations as *irrevocable* actions. For example, communicating partial results of a relaxed transaction to either the external world via an I/O operation or to other threads via a synchronization operation (such as a lock release or a write to a C++11 atomic variable) may constitute an irrevocable action because the system may not be able to roll back the effects that this communication had on the external world or other threads. For this reason, relaxed transactions cannot be cancelled (Section 8). Irrevocable actions may limit the concurrency in an implementation; for example, they may cause the implementation to not execute relaxed transactions concurrently with other transactions.

3.1 The `transaction_callable` function attribute

The `transaction_callable` attribute indicates that a function (including virtual functions and template functions) is intended to be called within a relaxed transaction. The `transaction_callable` attribute is intended for use by an implementation to improve the performance of relaxed transactions; for example, an implementation can generate a specialized version of a `transaction_callable` function, and execute that version when the function is called inside a relaxed transaction. Annotating a function with the `transaction_callable` attribute does not change the semantics of a program. In particular, a function need not be declared with the `transaction_callable` attribute to be called inside a relaxed transaction. Declaring a function with the `transaction_callable` attribute does not prevent the function from being called outside a relaxed transaction.

The `transaction_callable` attribute specifies a property of a specific function, not its type. It cannot be associated with pointers to functions, and may not be used in a `typedef` declaration.

A function declared with the `transaction_callable` attribute must not be re-declared without that attribute. A function declared without the `transaction_callable` attribute must not be re-declared with the `transaction_callable` attribute. See Section 10 for rules and restrictions on overriding virtual functions declared with the `transaction_callable` attribute.

3.2 Nesting

Relaxed transactions may be nested within other relaxed transactions.

```
// Starting value: x = 0, y = 0
int x = 0, y = 0;
__transaction_relaxed
{
   __transaction_relaxed
  {
      ++x;
  }
  ++y;
}
// Final value: x = 1, y = 1
```

3.3 Examples

The following example demonstrates the implementation of a swap operation using relaxed transactions. Note that Thread T2 cannot see the intermediate state where x == y from Thread T1.

int x = 1, y = 2;	
<u>Thread T1</u>	<u>Thread T2</u>

`__transaction_relaxed {`
`   int tmp = x;`
`   x = y;`
`   y = tmp;`
`}`
	`int tmpX = 0, tmpY = 0;`
	`__transaction_relaxed {`
	`   tmpX = x;`
	`   tmpY = y;`
	`}`
	`assert(tmpX != tmpY);`

The following example demonstrates how I/O can be used within relaxed transactions. The two output operations will not be interleaved between the relaxed transactions.

Output: "`Hello World.Hello World.`"	
<u>Thread T1</u>	<u>Thread T2</u>

`__transaction_relaxed {`
`   std:cout << "Hello World.";`
`}`
	`__transaction_relaxed {`
	`   std:cout << "Hello World.";`
	`}`

4. Atomic transactions

A transaction statement that uses the `__transaction_atomic` keyword defines an *atomic transaction*. We call such a statement an *atomic transaction statement:*

`__transaction_atomic` *compound-statement*

In a data-race-free program, an atomic transaction appears to execute atomically; that is, the compound statement appears to execute as a single indivisible operation whose operations do not interleave with the operations of other threads (Section 4.5). In this setting, atomic transactions allow a programmer to write code fragments that execute in isolation from other threads. The transactions do not observe changes made by other threads during their execution, and other threads do not observe partial results of the transactions.

An atomic transaction executes in an all-or-nothing fashion: it can be explicitly cancelled so that its operations have no effect (Section 8).

These properties make it easier to reason about the interaction of atomic transactions and the actions of other threads when compared to other synchronization mechanisms such as mutual exclusion.

To ensure that these guarantees can be made, atomic transactions are statically restricted to contain only "safe" code (Section 4.2). This ensures that an atomic transaction cannot execute code that would have visible side effects before the atomic transaction completes, such as performing certain synchronization and I/O operations. These same restrictions support the ability to cancel an atomic transaction explicitly by executing a cancel statement (Section 8), because they ensure that no visible side effects occur during the execution of the atomic transaction, and thus it is possible to roll back all changes made by an atomic transaction at any point during its execution.

4.1 Outer atomic transactions

A transaction statement annotated with the `outer` attribute defines an *outer atomic transaction*:

`__transaction_atomic [[ outer ]]` *compound-statement*

An outer atomic transaction is an atomic transaction that must not be nested lexically or dynamically within another atomic transaction. Thus, an outer atomic transaction must not appear within an atomic transaction or within the body of a function that might be called inside an atomic transaction (see Section 8.2) for details about how this is enforced).

Outer atomic transactions enable the use of the cancel-outer statement (Section 8.1), which can be executed only within the dynamic extent of an outer atomic transaction.

4.2 The `transaction_safe` and the `transaction_unsafe` attributes

To ensure that atomic transactions can be executed atomically, certain statements must not be executed within atomic transactions; we call such statements *unsafe*. (A statement is *safe* if it is not unsafe.) Because this restriction applies to the dynamic extent of atomic transactions, it must also apply to functions called within atomic transactions. To enable this restriction to be enforced, we distinguish between *transaction-safe* and *transaction-unsafe* function types. (There are also may-cancel-outer function types, as described in Section 8.2.)

Function declarations (including virtual and template function declarations), declarations of function pointers, and `typedef` declarations involving function pointers may specify `transaction_safe` or `transaction_unsafe` attributes. A function declared with the `transaction_safe` attribute has a transaction-safe type, and may be called within the dynamic extent of an atomic transaction. The `transaction_unsafe` attribute specifies a transaction-unsafe type. A transaction-safe type might also be specified by implicitly declaring a function safe, as described further in this section.

We sometimes abbreviate the statement that a function has transaction-safe or transaction-unsafe type by stating simply that the function is transaction-safe or transaction-unsafe, respectively.

A function type must not be both transaction-safe and transaction-unsafe. That is, function declarations, function pointer declarations, or `typedef` declarations for function pointer types must not specify both the `transaction_safe` and the `transaction_unsafe` attributes. If any declaration of such an entity specifies the `transaction_safe` attribute then every such declaration (except a function definition, if it is not a virtual function) must specify the `transaction_safe` attribute. A function declaration that specifies the `transaction_callable` attribute may also specify the `transaction_safe` or the `transaction_unsafe` attribute.

[Note: *A function declared in multiple compilation units must have the same type in all of these compilation units. For example, a function that has a transaction-safe type in one compilation unit must be declared to have such a type in all compilation units where it is declared.*]

Pointers to transaction-safe functions are implicitly convertible to pointers to the corresponding transaction-unsafe functions. Such conversions are treated as identity conversions for purposes of overload resolution, i.e., they have no effect on the ranking of conversion sequences. There is no conversion from transaction-unsafe function pointers to transaction-safe function pointers.

The `transaction_safe` and `transaction_unsafe` attributes specify properties of the type of the declared object, or of a type declared using `typedef`. Although such properties are ignored for overload resolution, they are part of the type and propagated as such. For example:

```
auto f = []()[[transaction_safe]] { g(); }
```

declares f() to be transaction-safe.

An atomic transaction or a body of a function declared with the `transaction_safe` attribute must not contain calls to transaction-unsafe functions and other unsafe statements, defined precisely below. This ensures that such statements are not executed within the dynamic extent of an atomic transaction.

A statement is *unsafe* if any of the following applies:

1. It is a relaxed transaction statement.
2. It is an atomic transaction statement annotated with the `outer` attribute (that is, it is an outer atomic transaction).
3. It contains an initialization of, assignment to, or a read from a volatile object.
4. It is an unsafe `asm` declaration; the definition of the unsafe `asm` declaration is implementation-defined.
5. It contains a function call to a function not known to have a transaction-safe or may-cancel-outer (Section 8.2) function type.

[Note: *A relaxed transaction is unsafe because it may contain unsafe statements (Section 3). An outer atomic transaction is unsafe because it cannot be nested within another atomic transaction. A statement that contains an initialization of, assignment to, or a read from a volatile object is unsafe because a value of a volatile object may be changed by means undetectable to an implementation. The definition of the unsafe `asm` declaration is implementation-defined because the meaning of the `asm` declaration is implementation-defined.*]

Although built-in operators are safe, they may be overloaded with user-defined operators, which result in function calls. Thus, applications of these operators may be safe or unsafe, as determined by the rules defined in this section. (For example, although the built-in `new` and `delete` operators are safe, user-defined `new` and `delete` operators may be unsafe. Atomic operations defined by the standard library are unsafe.)

A function definition *<u>implicitly</u> declares a function safe,* that is, declares its type to be transaction-safe, if the function is not a virtual function, its body contains only safe statements, and neither the definition nor any prior declaration of the function specifies any of the `transaction_unsafe`, `transaction_safe`, or `transaction_may_cancel_outer` (section 8.2) attributes. (If the definition or a prior declaration specifies the `transaction_safe` attribute, the function is of transaction-safe type, but the definition does not *implicitly* declare the function safe.) If the definition of a function implicitly declares it safe then no declaration of that function may specify the `transaction_unsafe` attribute. Note that a recursive function that directly calls itself is never implicitly declared safe. It may, however, explicitly specify a `transaction_safe` attribute.

A function template that does not specify any of the `transaction_safe`, `transaction_unsafe`, or `transaction_may_cancel_outer` attributes may define a template function that may or may not be implicitly declared safe, depending on whether the body of the template function contains unsafe statements after instantiation. (This feature is especially useful for template libraries, because it allows the use of template library functions within atomic transactions when they are instantiated to contain only safe statements, without requiring these template library functions to be always instantiated to contain only safe statements.) See Section 4.3 for an example of such a function template.

See Section 10 for rules and restrictions on overriding virtual functions declared with the `transaction_safe` attribute.

When a function pointer of transaction-safe type is assigned or initialized with a value, the initializing or right-hand-side expression must also have transaction-safe type. Furthermore, the transaction safety properties of function pointer parameter types must match exactly. In particular, the type of a function pointer parameter appearing in the type of the target pointer should be transaction-safe if and only if the corresponding parameter type in the initializing or right-hand-side expression is.

[Note: *An implementation may provide additional mechanisms that make statements safe. Such mechanisms might be necessary to implement system libraries that execute efficiently inside atomic transactions. Such mechanisms are intended for system library developers and are not part of this specification.*]

The creation (destruction) of an object implicitly invokes a constructor (destructor) function if the object is of a class type that defines a constructor (destructor). The constructor and destructor functions of a class must therefore have transaction-safe or may-cancel-outer type if the programmer intends to allow creation or destruction of objects of that class type inside atomic transactions. In the absence of appropriate programmer-defined constructors (destructors), the creation (destruction) of an object may implicitly invoke a compiler-generated constructor (destructor). A compiler-generated constructor (destructor) for a class has a transaction-safe type if the corresponding constructors (destructors) of all the direct base classes and the corresponding constructors (destructors) of all the non-static data members of the class have transaction-safe type. A compiler-generated constructor (destructor) for a class that is not derived from any other class and has no non-static members of class type always has transaction-safe type.

The assignment to an object invokes a compiler-generated assignment operator if the object belongs to a class that does not define an assignment operator. A compiler-generated assignment operator for a class has transaction-safe type if the corresponding assignment operators for all the direct base classes and the corresponding assignment operators for all the non-static data members of the class have transaction-safe type.

[Note: *The* `transaction_safe` *attribute on function and function pointer declarations allows the compiler to ensure that functions whose bodies contain unsafe statements are not called inside atomic transactions. Any function with external linkage that the programmer intends to be called* *inside atomic transactions in other translation units must be declared with the* `transaction_safe` *attribute. To allow client code to use libraries inside atomic transactions, library developers should identify functions with external linkage that are known and intended to contain only safe statements and annotate their declarations in header files with the* `transaction_safe` *attribute. Similarly, library developers should use the* `transaction_unsafe` *attribute on functions known or intended to contain unsafe statements. The* `transaction_unsafe` *attribute specifies explicitly in a function's interface that the function may contain unsafe actions and prevents a function from being implicitly declared safe so that future implementations of that function can contain unsafe statements. When annotating a function with the* `transaction_unsafe` *attribute, library developers should specify this attribute on both a function declaration and its definition when the declaration and the definition are located in separate header files. This enables client code to include such header files in an arbitrary order.*]

[Note: *Library users should not circumvent the restrictions imposed by the library interface by merely modifying transaction-related attributes in the library header files. Similar to other changes to a function declaration (such as changing a function return type or type of a function argument), adding, removing or modifying a transaction-related attribute requires re-compilation. Modifying transaction-related attributes in library header files without re-compiling the library may result in undefined behavior.*]

The header files for the C++ standard library should be modified to specify the annotations for the library functions consistent with the safety properties of those functions. Synchronization (that is, operations on locks and C++11 atomic operations) and certain I/O functions in the C++ standard library should not be declared to have transaction-safe type, as such actions could break atomicity of a transaction, that is, appear to interleave with actions of other threads, under the memory model rules specified in this document (Section 4.5).[2]

4.3 Examples

The following example shows a function declared transaction-safe via the `transaction_safe` attribute:

```
[[transaction_safe]] void f();
```

The following example shows a function implicitly declared safe by its definition:

```
int x;
void g()
{
   ++x; // body containing only safe statements
}
// g() is implicitly declared safe after this point
```

An atomic transaction can contain calls to functions declared transaction-safe either implicitly or by using an attribute, as illustrated by the following example:

```
void test()
{
     __transaction_atomic {
         f();  // OK because f() is declared transaction-safe using
               // the transaction_safe attribute
```

[2] We are currently investigating ways to partially overcome this limitation.

```
        g();   // OK because g() is implicitly declared safe
    }
}
```

The following example illustrates combinations of declarations:

```
void f(); // first declaration of f
void f() { ++w; } // OK, definition of f implicitly declares it transaction-safe
void f(); // OK, f is still declared transaction-safe

[[transaction_safe]] void g(); // first declaration of g
void g() { ++x;} // OK: transaction_safe attribute optional on definition
void g(); // Error: prior declaration has transaction_safe attribute

void h();  // first declaration of h
[[transaction_safe]] void h() {…} // Error: prior declaration has no
                                  // transaction_safe attribute

void k() { ++y;} // OK, first declaration of k is a definition that implicitly declares it safe
[[transaction_unsafe]] void k(); // Error: previous declaration of k
                                 // implicitly declared it safe

[[transaction_unsafe]] void l(); // first declaration of l
void l() { ++z;}; // OK, this definition does not implicitly declare k safe because of
                  // a prior declaration with the transaction_unsafe attribute

void m(); // first declaration of m
[[transaction_unsafe]] void m(); // OK, first declaration of m
                                 // did not declare it transaction-safe
```

The following example illustrates transaction-safe function pointers:

```
[[transaction_safe]] void (*p1)();
void (*p2)();
void foo();

p2 = p1;  // OK
p2 = f;   // OK
p1 = p2;  // Error: p2 is not transaction-safe
p1 = foo; // Error: foo is not transaction-safe
```

A programmer may instantiate function templates not declared with transaction-related attributes to form either transaction-safe or transaction-unsafe template functions, as shown in the following example:

```
template<class Op>
void t(int& x, Op f) { // Transaction-safety properties of t are not known at this point
      x++; f(x);
}

class A1 {
public:
    // A1::() is declared transaction-safe
    [[transaction_safe]] void operator()(int& x);
```

```
};

class A2 {
public:
     // A2::() is declared transaction-unsafe
     [[transaction_unsafe]] void operator()(int& x);
};

void n(int v) {
   __transaction_atomic {
      t(v, A1());                        // OK, call to t<A1> is safe
      t(v, A2());                        // Error, call to t<A2> is unsafe
   }
}
```

The following example illustrates using template functions with function pointer or lambda expression arguments:

```
[[transaction_safe]] void (*p1) (int&);
void (*p2) (int&);
[[transaction_unsafe]] void u();

void n(int v) {
   int total = 0;
   __transaction_atomic {
      t(v, p1);                          // OK,  the call is safe
      t(v, [&](int x) {total += x;}); // OK, the call is safe
      t(v, p2);                          // Error,  the call is unsafe
      t(v, [&](int x) {u();});           // Error, the call is unsafe
   }
}
```

4.4 Nesting

Atomic transactions except outer atomic transactions are safe statements and thus may be nested lexically (i.e., an atomic transaction may contain another atomic transaction) or dynamically (i.e., an atomic transaction may call a function that contains an atomic transaction).

The following example shows an atomic transaction lexically nested within another atomic transaction:

```
__transaction_atomic {
    x++;
    __transaction_atomic {
        y++;
    }
    z++;
}
```

The following example shows an atomic transaction dynamically nested within another atomic transaction:

```
[[ transaction_safe ]] void bar()
{
   __transaction_atomic { x++; }
}
```

```
__transaction_atomic {
   bar();
}
```

Atomic transactions may be nested within relaxed transactions. Relaxed transactions must not be nested within atomic transactions (Section 4.2).

4.5 Memory model

The memory model rules for transactions (Section 2.1) are sufficient to guarantee that in data-race-free programs, atomic transactions appear to execute as single indivisible operations. This is ensured by restricting atomic transactions so that they do not contain other forms of synchronization, such as, operations on locks or C++11 atomic operations (Section 4.2). Consequently, an operation executed by one thread cannot be ordered by the "happens-before" relationship between the StartTransaction and EndTransaction operations of an atomic transaction by another thread, and thus cannot appear to interleave with operations of an atomic transaction executed by another thread.

5. Transaction expressions

The `__transaction_relaxed` or `__transaction_atomic` keyword followed by a parenthesized expression defines a *transaction expression*. Unlike a transaction statement, a transaction expression defined by the `__transaction_atomic` keyword must not be annotated with the `outer` attribute:

`__transaction_relaxed (` *expression* `)`
`__transaction_atomic (` *expression* `)`

A transaction expression of type T is evaluated as if it appeared as a right-hand side of an assignment operator inside a transaction statement:

`__transaction_atomic {` T temp = *expression* `;}`

The value of the transaction expression is the value of a variable temp in the left-hand side of the assignment operator. If T is a class type, then variable temp is treated as a temporary object.

A transaction expression can be used to evaluate an expression in a transaction. This is especially useful for initializers, as illustrated by the following example:

`SomeObj myObj = __transaction_atomic (` *expr* `); //` calls copy constructor

In this example a transaction expression is used to evaluate an argument of a copy constructor in a transaction. This example cannot be expressed using just transaction statements because enclosing the assignment statement in a transaction statement would restrict the scope of the `myObj` declaration.

[Note: *A transaction expression on an initializer applies only to evaluating the initializer. The initialization (for example, executing a copy constructor) is performed outside of a transaction. Transaction expressions and statements thus do not allow a programmer to specify that the initialization statement should be executed inside a transaction without restricting the scope of the initialized object.*]

A transaction expression cannot contain a transaction statement, a cancel statement (Section 8) or a cancel-and-throw statement (Section 9) since the C++ standard does not allow expressions to contain statements.

Implementations that support statement-expressions could syntactically allow a cancel statement or a cancel-and-throw statement to appear within a transaction expression. However, a cancel or cancel-and-throw statement must not appear inside a transaction expression unless the cancel or cancel-and-throw statement is either annotated with the `outer` attribute or is lexically enclosed within an atomic transaction statement that is lexically enclosed within that transaction expression.

6. Function transaction blocks

The function transaction block syntax specifies that a function's body – and, in the case of constructors, all member and base class initializers – execute inside a transaction; for example:

```
void f() __transaction_relaxed {
   // body of f() executes in an relaxed transaction
}

void g() __transaction_atomic {
   // body of g() executes in a atomic transaction
}
```

Like a transaction expression, a function transaction block may not be annotated the `outer` attribute.

A function transaction block on a constructor causes the constructor body and all member and base class initializers of that constructor to execute inside a transaction. The function transaction block syntax thus allows programmers to include member and base class initializers in constructors in a transaction. In the following example, the constructor `Derived()` and its initializers all execute atomically:

```
class Base {
public:
    Base(int id) : id_(id) {}
private:
    const int id_;
};

class Derived : public Base {
public:
   Derived() __transaction_atomic : Base(count++) { … }
private:
   static int count = 0;
};
```

This example shows a common pattern in which each newly allocated object is assigned an id from a global count of allocated elements. This example cannot be expressed using just transaction statements: the static field count is shared so it must be incremented inside some form of synchronization, such as an atomic transaction, to avoid data races. But the field id_ is a const member of the base class and can be initialized only inside the base class constructor, which in turn can be initialized only via a member initializer list in the derived class.

A function transaction block can be combined with the function try block syntax. If the `__transaction_atomic` or the `__transaction_relaxed` keyword appears before the `try` keyword, the catch block is part of the function transaction block. If the transaction keyword appears after the `try` keyword, the catch block is not part of the function transaction block:

```
Derived::Derived()
try __transaction_atomic : Base(count++) {}
catch (...) {} // catch is not part of transaction

Derived::Derived()
__transaction_atomic try : Base(count++) { … }
catch (...) {} // catch is part of transaction
```

[Note: *A function with a function transaction block may be declared with a transaction-related attribute (i.e., `transaction_safe`, `transaction_unsafe`, `transaction_callable`, or `transaction_may_cancel_outer` (Section 8.2)). The legality of such combinations is governed by general rules of this specification. For example, the following code is erroneous, as a relaxed function transaction block (unsafe statement) cannot occur in a function declared with the `transaction_safe` attribute:*

```
// error: a relaxed transaction is never transaction-safe
[[transaction_safe]] void f() __transaction_relaxed { … }
```
]

Unlike a transaction statement, a function transaction block may contain a cancel statement only if that cancel statement is annotated with the `outer` attribute or is enclosed by an atomic statement nested inside the function transaction block (Section 8). A function transaction block may contain a cancel-and-throw statement (Section 9).

7. Noexcept specification[3]

The body of a transaction statement (expression) may throw an exception that is not handled inside its body and thus propagates out of the transaction statement (expression).

Transaction statements and expressions may have noexcept specifications that explicitly state if exceptions may or may not be thrown by the statement:

`__transaction_atomic noexcept [(`*constant-expression*`)]` *compound-statement*
`__transaction_atomic noexcept [(`*constant-expression*`)] (` *expression* `)`

`__transaction_relaxed noexcept [(`*constant-expression*`)]` *compound-statement*
`__transaction_relaxed noexcept [(`*constant-expression*`)] (` *expression* `)`

Transaction statements and expressions that use noexcept specifications may be annotated with an attribute, which should appear between the `__transaction_atomic` or `__transaction_relaxed` keyword and the noexcept operator, as illustrated by the following example:

`__transaction_atomic [[ outer ]] noexcept`
`[(` *constant-expression* `)]` *compound-statement*

The noexcept clause without a constant-expression or with a constant-expression that evaluates to `true` indicates that a transaction statement (expression) must not throw an exception that escapes the scope of the transaction statement (expression). Throwing an exception that

[3] Previous versions of this specification included rules that enabled the use of exception specifications with transactions statements. Because C++11 has deprecated exception specifications, we have since removed them and replaced them with noexcept specifications, which are new to C++11. With this change, a transaction statement may now only specify that no exceptions can escape its scope or all can.

escapes the scope of the transaction statement in this case results in a call to `std::terminate()`.

The following example declares a transaction statement that does not allow an exception to propagate outside of its scope:

`__transaction_atomic noexcept (true)` *compound-statement*
`__transaction_atomic noexcept` *compound-statement*

A transaction statement (expression) that does not include a noexcept specification or includes a noexcept specification that has a constant-expression that evaluates to `false` may throw an exception that escapes the scope of the transaction statement (expression).

The following example declares a transaction statement that allows an exception to propagate outside of its scope:

`__transaction_atomic noexcept(false)` *compound-statement*
`__transaction_atomic` *compound-statement*

[Note: *Omitting a noexcept specification on a* transaction *statement (expression) that may throw an exception makes it easy to overlook the possibility that an exception thrown from within the dynamic extent of that statement (expression) can result in the statement (expression) being only partially executed. Therefore, programmers are strongly encouraged to explicitly state whether exceptions can be thrown from transaction statements (expressions) by using noexcept specifications. We considered an alternative approach in which the absence of a noexcept specification is interpreted as if a* `noexcept(true)` *clause were present, which makes mandatory an explicit noexcept(false) specification on a transaction statement (expression) that may throw an exception. However, such an interpretation would be inconsistent with the existing rules for noexcept specifications on function declarations.*]

A noexcept specification is not allowed on a function transaction block as such a specification is redundant with a noexcept specification on a function declaration (that is, a noexcept specification that may appear before the `__transaction_atomic` or `__transaction_relaxed` keyword denoting a function transaction block).

8. Cancel statement

The `__transaction_cancel` statement (a *cancel statement*) allows the programmer to roll back an atomic transaction statement. The cancel statement must be lexically enclosed in an atomic transaction statement, unless it is annotated with the `outer` attribute (Section 8.1); for example:

```
__transaction_atomic {
    stmt1
    __transaction_cancel;
}
stmt2
```

In its basic form (that is, without the `outer` attribute), a cancel statement rolls back all side effects of the immediately enclosing atomic transaction statement (that is, the smallest atomic transaction statement that encloses the cancel statement) and transfers control to the statement following the transaction statement. Thus, in the example above the cancel statement undoes the side effects of *stmt1* and transfers control to *stmt2*.

The rule requiring a cancel statement to be lexically enclosed in an atomic transaction statement ensures that the cancel statement always executes within the dynamic extent of an atomic transaction statement. It also allows the implementation to distinguish easily between atomic transactions that require rollback and those that don't, a potential optimization opportunity for an implementation.

[Note: *A cancel statement applies only to atomic transaction statements (including outer atomic transaction statements). A cancel statement cannot be used to roll back a function transaction block or a transaction expression, unless that block or expression is rolled back as part of rolling back an atomic transaction statement.*]

8.1 The `outer` attribute on cancel statements

Cancel statements may be annotated with the `outer` attribute:

```
__transaction_cancel [[ outer ]];
```

We call a cancel statement with the `outer` attribute a *cancel-outer* statement. A cancel-outer statement rolls back all side effects of the outer atomic transaction that dynamically contains it (which is also the outermost atomic transaction that dynamically contains it) and transfers control to the statement following the outer atomic transaction.

Unlike a cancel statement with no attribute, a cancel-outer statement need not be enclosed within the lexical scope of an atomic transaction. Instead, to ensure that a cancel-outer statement always executes within the dynamic extent of an outer atomic transaction, a cancel-outer statement must appear either within the lexical scope of an outer atomic transaction or in a function declared with the `transaction_may_cancel_outer` attribute (Section 8.2).

[Note: *A cancel-outer statement cancels only outer atomic transactions; the restrictions above imply that a cancel-outer statement cannot be executed when the outermost atomic transaction is not an outer atomic transaction. In contrast, an unannotated cancel statement can cancel an outer atomic transaction if it is the immediately enclosing atomic transaction.*]

The cancel-outer statement provides a convenient way to cancel an outermost atomic transaction from anywhere within its dynamic extent. For example, when an error is encountered, the programmer can cancel all transactions from the most nested transaction to the outer transaction. The outer atomic transaction – together with the `transaction_may_cancel_outer` attribute – ensures that an outermost atomic transaction that may dynamically contain a cancel-outer statement is easily identifiable as such. This is important because otherwise, it would be difficult to determine whether a given atomic transaction might be cancelled without examining all code it might call.

[Note: *Cancelling an outermost atomic transaction using either multiple cancel statements without the `outer` attribute or exceptions both have the disadvantages that additional, error-prone code would be required to transfer control back to the outermost atomic transaction and to cancel the outermost atomic transaction.*]

8.2 The `transaction_may_cancel_outer` attribute

Function declarations (including virtual and template function declarations) and function pointer declarations may specify the `transaction_may_cancel_outer` attribute. The `transaction_may_cancel_outer` attribute specifies that the declared function (or a function pointed to by the declared function pointer) has may-cancel-outer type and hence may contain a cancel-outer statement within its dynamic scope. Like cancel-outer statements, a call to a function with may-cancel-outer type must appear either within the lexical scope of an outer atomic transaction or in a may-cancel-outer function.

If a class type has constructors or a destructor with may-cancel-outer type, then objects of that type must be declared so as to ensure that the affected constructor or destructor is invoked within the dynamic scope of an outer atomic transaction. Declarations of such an object leading to the invocation of the affected constructor or destructor should appear within the lexical scope of an outer atomic transaction or in a may-cancel-outer function. Moreover, an object should be declared in such a way that the affected constructor or destructor is invoked in the same scope as the declaration. For example, if a class has a constructor with may-cancel-outer type then a program may not contain global or static declarations of that type resulting in the invocation of the affected constructor. If a class has a destructor with may-cancel-outer type then a program may not contain global, static, function-local static or thread_local declarations of that type.

Like the `transaction_safe` attribute, the `transaction_may_cancel_outer` attribute specifies a property of the type of the declared function or function pointer, which is propagated along with the type. Just like `transaction_safe`, it may be meaningfully used in `typedef` declarations.

A pointer to a function with transaction-safe type may be implicitly converted to a pointer to a function with may-cancel-outer type (or to a pointer to a function with transaction-unsafe type). Such conversions have no effect on the ranking of conversions sequences. A pointer to a may-cancel-outer function is not implicitly convertible to a pointer to a non-may-cancel-outer function. Allowable function pointer conversions:

transaction-safe → may-cancel-outer
transaction-safe → transaction-unsafe

A function or function pointer must not be declared with both the `transaction_may_cancel_outer` and `transaction_safe` attributes. A function must not be declared with both the `transaction_may_cancel_outer` and `transaction_unsafe` attributes. That is, a function declaration, a function pointer declaration, or multiple declarations of one function must not specify both attributes. If any declaration of a function specifies the `transaction_may_cancel_outer` attribute then every declaration of that function (except its definition, if it is not a virtual function) must specify the `transaction_may_cancel_outer` attribute, and the first declaration must do so even if it is the definition of a non-virtual function. The `main` function must not be declared with the `transaction_may_cancel_outer` attribute. A function may be declared with both `transaction_may_cancel_outer` and `transaction_callable` attributes.

A function call to a function declared with the `transaction_may_cancel_outer` attribute (before the function call) is a safe statement (Section 4.2). A function call through a function pointer that was declared with the `transaction_may_cancel_outer` attribute is also a safe statement. The body of a function declared with the `transaction_may_cancel_outer` attribute must not contain unsafe statements.

See Section 10 for rules and restrictions on overriding virtual functions declared with the `transaction_may_cancel_outer` attribute.

When a function pointer declared with the `transaction_may_cancel_outer` attribute is assigned or initialized with a value, that value must be a pointer to a function of transaction-safe or may-cancel-outer type. When a function pointer declared without the `transaction_may_cancel_outer` attribute is assigned or initialized with a value, that value must not be a pointer to a function of may-cancel-outer type. As in the case of the `transaction_safe` attribute, parameter types for function pointers assigned in this way must match exactly in their `transaction_may_cancel_outer` specification.

8.3 Examples

An unannotated cancel statement rolls back the side effects of only its immediately enclosing atomic transaction. In the following example, the cancel statement rolls back stmt2 but not stmt1.

```
bool flag1 = false, flag2 = false;
__transaction_atomic {
    flag1 = true; // stmt1
    __transaction_atomic {
        flag2 = true; // stmt2
        __transaction_cancel;
    }
    assert (flag1 == true && flag2 == false);
}
assert (flag1 == true && flag2 == false);
```

A cancel-outer statement rolls back the side effects of the outer atomic transaction that dynamically contains it. In the following example, the cancel-outer statement rolls back both stmt2 and stmt1.

```
bool flag1 = false, flag2 = false;
__transaction_atomic [[outer]] {
    flag1 = true; // stmt1
    __transaction_atomic {
        flag2 = true; // stmt2
        __transaction_cancel [[outer]];
    }
    assert (0); // never reached!
}
assert (flag1 == false && flag2 == false);
```

A cancel statement may execute within a dynamic scope of a relaxed transaction. The following example shows an “atomic-within-relaxed” idiom that dynamically combines cancelling a transaction and irrevocable actions within a relaxed transaction:

```
[[transaction_safe]] void do_work();
[[transaction_safe]] bool all_is_ok();
[[transaction_unsafe]] void report_results(); // contains irrevocable actions

__transaction_relaxed {
    bool all_ok = false;
    __transaction_atomic {
        do_work();
        if (all_is_ok())
           all_ok = true;
        else
           __transaction_cancel;
    }
    if (all_ok)
       report_results();
}
```

8.4 Memory model

Cancelling an atomic transaction removes all side effects of its execution. Consequently, in a data-race-free program a cancelled atomic transaction has no visible side effects. Cancelling an atomic transaction, however, does not remove a data race that occurred during the execution of the transaction. The individual operations of an atomic transaction that executed before the transaction was cancelled are part of the program execution and, like other operations, may contribute to data races. In case of a data race, the program behavior is still undefined, as specified by the C++11 memory model. For example, the following program is deemed racy even though the transaction with a racy memory access is cancelled:

Thread 1	Thread 2

`__transaction_atomic {`
`    x++;`
`    __transaction_cancel;`
`}`
	`x = 1;`

9. Cancel-and-throw statement

A programmer can use a *cancel-and-throw* statement to rollback all side effects of an atomic transaction statement (atomic function transaction block) and cause that statement (block) to throw a specified exception. The cancel-and-throw statement must be lexically enclosed in an atomic transaction statement (atomic function transaction block), unless it is annotated with the `outer` attribute (Section 9.1); for example:

```
__transaction_atomic {
   stmt1
   __transaction_cancel throw throw-expression;
}
```

In its basic form (that is, without the `outer` attribute), the cancel-and-throw statement rolls back all side effects of the immediately enclosing atomic transaction statement (atomic function transaction block) and throws the exception from the transaction. Thus, in the example above the cancel-and-throw statement undoes the side effects of *stmt1* and throws *`throw-expression`*.

The exception thrown by the cancel-and-throw statement must be of integral or enumerated type. This restriction ensures that the exception does not contain or refer to state that is not meaningful after the transaction is cancelled.[4]

[Note: *The programmer should not circumvent the restriction on the exception types by using the exception, for example, as an index in a global array that stores additional information about the exception. Since the exception will be processed in an environment in which the memory effects of the transaction have been rolled back, code like the following may compile, but is never useful:*

```
__transaction_atomic {
      int my_exc_index = doSomething();
      if (my_exc_index >= 0) {
         real_exception_description[my_exc_index] =
            new( <detailed information about exception> );
         __transaction_cancel throw my_exc_index;
      }
}]
```

The exception thrown by a cancel-and-throw statement will not be caught by any try-catch block nested within the cancelled atomic transaction.

[4] Section “Removing restrictions on types of exceptions thrown by the cancel-and-throw statement” in Appendix C explains the rationale for this restriction in more detail.

In an exception handler of integral or enumerated type, the cancel-and-throw statement may optionally leave out the exception expression, in which case the specified exception is the current exception.

A cancel-and-throw statement has the same properties with respect to the memory model as a cancel statement (Section 8.4): In a data-race-free program, a transaction cancelled by a cancel-and-throw statement has no visible side effects. However, the individual operations of a transaction that executed before the transaction was cancelled are part of the program execution and may contribute to data races.

Unlike a regular throw statement, a cancel-and-throw statement provides strong exception safety guarantees. With a regular throw statement, it is the programmer's responsibility to restore the invariants that might be violated by partial execution of an atomic transaction. With a cancel-and-throw statement the system automatically guarantees that such invariants are preserved by rolling back the atomic transaction.

9.1 The outer attribute on cancel-and-throw statements

The cancel-and-throw statement may be annotated with the `outer` attribute, in which case it is a *cancel-outer-and-throw* statement:

```
__transaction_cancel [[ outer ]] throw expr_opt;
```

A cancel-outer-and-throw statement operates in the same way as a cancel-and-throw statement except that it rolls back the side effects of the outer atomic transaction that dynamically contains it and throws the exception from the outer atomic transaction. Like the cancel-outer statement, a cancel-outer-and-throw statement need not be enclosed within the lexical scope of an atomic transaction, but it must appear either within the lexical scope of an outer atomic transaction or in a may-cancel-outer function.

9.2 Examples

An unannotated cancel-and-throw statement rolls back the side effects of only its immediately enclosing atomic transaction. In the following example, the cancel-and-throw statement rolls back *stmt2* but not *stmt1*, and the thrown exception `1` propagates out of the outermost atomic transaction:

```
bool flag1 = false, flag2 = false;
try {
    __transaction_atomic {
        flag1 = true; // stmt1
        __transaction_atomic {
          flag2 = true; // stmt2
          __transaction_cancel throw 1;
      }
    }
} catch(int& e) {
    assert(flag1 == true && flag2 == false);
}
```

A cancel-outer-and-throw statement rolls back the side effects of the outer atomic transaction that dynamically contains it. In the following example, the cancel-outer-and-throw statement rolls back both *stmt1* and *stmt2*, after which the thrown exception `1` propagates out of the outer atomic transaction (which is the outermost atomic transaction):

```
bool flag1 = false, flag2 = false;
try {
    __transaction_atomic [[outer]] {
        flag1 = true; // stmt1
        __transaction_atomic {
          flag2 = true; // stmt2
          __transaction_cancel [[outer]] throw 1;
      }
    }
} catch(int& e) {
    assert(flag1 == false && flag2 == false);
}
```

The exception thrown by a cancel-and-throw statement cannot be caught by any try-catch block nested within the cancelled atomic transaction. In the following examples, Example 1 demonstrates how normal C++ try / catch blocks behaves inside a transaction, followed by Example 2, which demonstrates how a `__transaction_cancel` behaves inside a transaction. Notice that in Example 2 the first catch block does not catch the exception thrown by the cancel-and-throw:

Example 1:

```
try {
    __transaction_atomic {
        try {
            throw 1;
        } catch(int& e) {
            ... ; // exception is caught here
        }
    }
} catch (int& e) {
    assert(0); // never reached!
}
```

Example 2:

```
try {
    __transaction_atomic {
        try {
            __transaction_cancel throw 1;
        } catch(int& e) {
            assert(0); // never reached!
        }
    }
} catch (int& e) {
    cout << "Caught e!" << endl;
}
```

An exception thrown by a cancel-and-throw statement must be of integral or enumerated type. In the following example, the cancel-and-throw statement with the exception expression of type X is illegal:

```
class X { int x;};

__transaction_atomic noexcept(false) {
```

```
    __transaction_cancel throw X(); // Error: X() is of class type
}
```

A cancel-and-throw statement without an exception expression re-throws the current exception. In the following example, any exception thrown by *stmt* cancels the atomic transaction and propagates to a catch block higher up the stack:

```
__transaction_atomic noexcept(false) {
    try {
        stmt
    } catch (int&) {
        __transaction_cancel throw;
    }
}
```

A cancel-and-throw statement without an exception expression must occur within an exception handler of integral or enumerated type. In the following example, the cancel-and-throw statement is illegal because it occurs within an exception handler that matches any exception:

```
__transaction_atomic noexcept(false) {
    try {
        stmt
    } catch (...) {
        __transaction_cancel throw;  // Error: current exception may be of any type
    }
}
```

10. Inheritance and compatibility rules for attributes

A member function declared with a transaction-related attribute (i.e., `transaction_safe`, `transaction_unsafe`, `transaction_callable`, or `transaction_may_cancel_outer` attribute) in a base class preserves that attribute in the derived class unless it is redefined or overridden by a function with a different attribute. Functions brought into the class via a `using` declaration preserve the attributes that they had in their original scope. Transaction-related attributes impose no restrictions on redefining a function in a derived class. Transaction-related attributes impose the following restrictions on overriding a virtual function in a derived class:

- A virtual function of transaction-safe type may be overridden only by a virtual function of transaction-safe type.
- A virtual function of may-cancel-outer type can be overridden only by a virtual function of either may-cancel-outer or transaction-safe type.
- A virtual function of may-cancel-outer type may override only a virtual function of may-cancel-outer type.
- Any function pointer type appearing in a signature of an overriding function must have the same transactional attributes as the corresponding function pointer type in the signature of the overridden function.

The following example illustrates the class inheritance rules for transaction-related function attributes:

```
class C {
public:
    [[transaction_safe]] void f();
    [[transaction_safe]] virtual void v();
    [[transaction_unsafe]] virtual void w();
};
```

```
class D : public C {
public:
  void f();                // OK: D::f  redefines C::f
  virtual void v();        // Error: D::v overrides C::v; needs transaction_safe
  virtual void w();        // OK:  transaction_unsafe on  D::w is optional
  using C::v;              // OK: C::v preserves the transaction_safe attribute
};
```

11. Class attributes

The `transaction_safe`, `transaction_unsafe`, and `transaction_callable` attributes can be used on classes and template classes. In this case they act as default attributes for the member functions declared within the (template) class but not for member functions on any inheriting class; that is, they are applied to only those member functions declared within the (template) class that do not have an explicit `transaction_safe`, `transaction_unsafe`, `transaction_may_cancel_outer`, or `transaction_callable` attribute. The class attribute does not apply to functions brought into the class via inheritance or via a `using` declaration; such functions preserve the attributes that they had in their original scope.

The following example shows a definition of class C from Section 10 written using class attributes:

```
class C [[transaction_safe]] {
    void f();                          // declared as transaction_safe
    virtual void v();                  // declared as transaction_safe
    [[transaction_unsafe]] virtual void w(); // declared as
                                             // transaction_unsafe
};
```

Class attributes reduce C++ programming overhead as they allow the programmer to specify an attribute once at the class level rather than specifying it for each member function. We felt it was important to ease the programmer's task of specifying attributes to make them usable.

Appendix A. Grammar

atomic transaction-statement:
`__transaction_atomic` *txn-outer-attribute*$_{opt}$ *txn-noexcept-spec*$_{opt}$ *compound-statement*

relaxed transaction-statement:
`__transaction_relaxed` *txn-noexcept-spec*$_{opt}$ *compound-statement*

atomic transaction-expression:
`__transaction_atomic` *txn-noexcept-spec*$_{opt}$ `(` *expression* `)`

relaxed transaction-expression:
`__transaction_relaxed` *txn-noexcept-spec*$_{opt}$ `(` *expression* `)`

atomic function-transaction-block:
atomic-basic-function-transaction-block
`__transaction_atomic` *basic-function-try-block*

relaxed function-transaction-block:
relaxed basic-function-transaction-block
`__transaction_relaxed` *basic-function-try-block*

atomic basic-function-transaction-block
`__transaction_atomic` *ctor-initializer*$_{opt}$ *compound-statement*

relaxed basic-function-transaction-block
`__transaction_relaxed` *ctor-initializer*$_{opt}$ *compound-statement*

cancel-statement:
`__transaction_cancel` *txn-outer-attribute*$_{opt}$ `;`

cancel-and-throw-statement:
`__transaction_cancel` *txn-outer-attribute*$_{opt}$ *throw-expression* `;`

txn-noexcept-spec:
noexcept-specification

txn-outer-attribute:
`[[ outer ]]`

postfix-expression:
/ … existing C++11 rules …*/*
transaction-expression

statement:
  / … existing C++11 rules …*/*
  attribute-specifier$_{opt}$ transaction-statement

jump-statement:
  / … existing C++11 rules …*/*
  cancel-statement
  cancel-and-throw-statement

function-body:
  / … existing C++11 rules …*/*
  function-transaction-block

function-try-block:
  basic-function-try-block
  `try` *basic-function-transaction-block handler-seq*

basic-function-try-block:
  / … existing C++11 rules for function-try-block …*/*

Appendix B. Feature dependences

In this section, we identify the dependences between features, to assist implementers who might be considering implementing subsets of the features described in this specification or enabling features in different orders, dependent on implementation-specific tradeoffs.

As general guidance, we recommend that an implementation that does not support a certain feature accepts the syntax of that feature and issues an informative error message, preferably indicating that the feature is not supported by the implementation but is a part of the specification.

The language features described in this specification are interdependent. Eliminating a certain feature may make some other features unusable. For example, without the outer atomic transactions, the cancel-outer statement is unusable; that is, it is not possible to write a legal program that executes a cancel-outer statement and does not contain an outer atomic transaction statement (because the cancel-outer statement must execute within the dynamic extent of an outer atomic transaction). Some other features may remain usable but become irrelevant. For example, without atomic transactions, the `transaction_safe` attribute can occur in legal programs but serves no purpose. We recommend that an implementation that chooses to support a certain irrelevant feature issues an informative warning specifying that the feature is supported for compatibility purposes but has no effect. In the rest of this section, we describe dependences between the features and identify the consequences of omitting a particular feature or combination of features.

Transaction statements, transaction expressions and function transaction blocks. This specification provides three language constructs for specifying transactions: transaction statements, transaction expressions and function transaction blocks. All other features described in this specification are dependent on the presence of at least one of these constructs. Therefore any implementation should include at least one of these constructs. The constructs themselves are independent of each other. An implementation may include one, two or all three of them.

All three constructs allow for specifying two forms of transactions – relaxed transactions and atomic transactions. Furthermore, atomic statements may be annotated with the `outer` attribute to indicate that they execute as outer atomic transactions. These forms of transactions are independent of each other. An implementation may include either relaxed transactions, or atomic transactions, or both. It may also choose not to support outer atomic transactions, or to require all atomic transactions to be outer atomic transactions.

A majority of the features described in this specification are used in conjunction with atomic transactions. Eliminating or limiting support for atomic transactions makes many other features either unusable or irrelevant:

- The concept of safe and unsafe statements and the `transaction_safe` and `transaction_unsafe` function attributes are irrelevant without atomic transactions (because the safety concept and attributes are used to impose restrictions on statements that can be executed within an atomic transaction).
- The cancel statement is unusable without atomic transaction statements (because it applies only to atomic transaction statements).
- The cancel-and-throw statement is unusable unless an implementation supports either atomic transaction statements or atomic function transaction blocks (because it applies only to atomic transaction statements or atomic function transaction blocks).
- The cancel-outer statement, the cancel-outer-and-throw statement, and the `transaction_may_cancel_outer` attribute are unusable without outer atomic transactions (because the cancel-outer statements, cancel-outer-and-throw statements and calls to functions declared with the `transaction_may_cancel_outer` attribute can execute only within the dynamic extent of an outer atomic transaction).

The only feature used solely in conjunction with relaxed transactions is the `transaction_callable` attribute. This attribute is irrelevant without relaxed transactions (because it indicates that a function might be called within a relaxed transaction).

An implementation may impose additional restrictions on nesting of various forms of transactions without affecting the rest of the specified features.

Function call safety. This specification includes three features related to the safety of function calls – the `transaction_safe` and `transaction_unsafe` attributes and the concept of functions being implicitly declared safe. Eliminating one or more of the function call safety features does not affect the rest of the specification. However, different combinations of these features offer different degrees of ability to call functions from within atomic transactions:

- An implementation that does not support either the `transaction_safe` attribute or the concept of functions being implicitly declared safe must disallow function calls inside atomic transactions (because it has no ability to verify that such function calls are safe). In such an implementation, the `transaction_unsafe` attribute is irrelevant, as there is no way for a function to be declared safe.
- An implementation that supports functions being implicitly declared safe but does not support the `transaction_safe` attribute limits function calls inside atomic transactions to calling functions defined within the same translation unit before the transaction.
- An implementation that does not support functions being implicitly declared safe does not allow a function to be used in a transaction unless it is explicitly annotated with the `transaction_safe` attribute. For example, this prevents the use of a template library function that cannot be annotated with the `transaction_safe` attribute because it can only be determined to be safe after instantiation.
- If an implementation does not support the `transaction_unsafe` attribute, programmers cannot override the `transaction_safe` class attribute or prevent functions from being implicitly declared safe when this is not desirable. The first limitation is relevant if class attributes and the `transaction_safe` attribute are supported; the second limitation is relevant if functions can be implicitly declared safe.

An implementation may include the `transaction_safe` attribute for function declarations, or function pointer declarations, or both. An implementation that does not support the `transaction_safe` attribute for function pointer declarations must disallow calls via function pointers inside atomic transactions.

Cancel and cancel-and-throw statements. This specification provides two forms of a cancel statement – a basic cancel statement that cancels the immediately enclosing atomic transaction and the cancel-outer statement that cancels the enclosing outer atomic transaction. This specification also provides two similar forms of a cancel-and-throw statement – a basic cancel-and-throw statement and the cancel-and-throw-outer statement. The cancel and cancel-and-throw statements and the two forms of each statement are independent of each other. An implementation may include any combination of these statements and their forms. Eliminating either the basic cancel statement or the basic cancel-and-throw statement does not affect the rest of the specification. Eliminating either the cancel-outer statement or the cancel-outer-and-throw statement, but not both of these statements, also does not affect the rest of the features. Eliminating both the cancel-outer statement and the cancel-outer-and-throw statement makes the `transaction_may_cancel_outer` attribute irrelevant (because this attribute is used to specify that a function may contain either the cancel-outer or cancel-outer-and-throw statement in its dynamic scope) and limits the usability of the `outer` attribute on transaction statements (because the main purpose of this attribute is to specify atomic transactions that can be cancelled by the cancel-outer or cancel-outer-and-throw statement). The `outer` attribute, however, still can be used to specify that an atomic transaction statement cannot be nested within another atomic transaction.

The `transaction_may_cancel_outer` attribute. Eliminating the `transaction_may_cancel_outer` attribute reduces the usability of the cancel-outer and cancel-outer-and-throw statements. An implementation that does not support this attribute must not allow the cancel-outer and the cancel-outer-and-throw statements outside of the lexical scope of an outer atomic transaction statement (because the implementation has no ability to verify that a function containing a cancel-outer statement in its dynamic scope is not called outside of an outer atomic transaction).

The `transaction_callable` attribute. This attribute has no semantic meaning: it is only a hint to the compiler that certain optimizations might be worthwhile. Eliminating this attribute has no effect on other features.

noexcept specification. A noexcept specification facilitates development of more reliable programs. Not supporting noexcept specifications on transaction statements and/or expressions has no effect on other features.

Exceptions. An implementation that implements a subset of this specification may choose to provide limited support for exceptions inside transactions (including the exceptions thrown by the throw statement and/or exceptions thrown by the cancel-and-throw statement). For example, an implementation might disallow throwing an exception from within code that could be executed within a transaction, or disallow exceptions from escaping the scope of a transaction. Such restrictions might make noexcept specifications irrelevant.

Unsafe statements. This specification defines certain statements as unsafe. An implementation that implements a subset of this specification might choose to treat additional statements as unsafe. For example, an implementation might choose to treat built-in `new` and `delete` operators as unsafe and disallow them inside atomic transactions. We suggest that such an implementation provides a workaround to allow programmers to allocate and deallocate objects within atomic transactions, and indicate this in an error message produced when encountering a `new` or `delete` built-in operator in an atomic transaction. In most cases, treating additional statements as unsafe would not affect the rest of the specification.

Class attributes. Class attributes have no semantic meaning: they are default attributes for function members declared without a transaction-related attribute. Eliminating class attributes has no effect on the rest of the features.

Appendix C. Extensions

Allowing unsafe statements inside atomic transactions. To relax the restriction of statically disallowing unsafe statements inside atomic transactions and functions declared with the `transaction_safe` or `transaction_may_cancel_outer` attribute, we could make executing such statements a dynamic error that rolls back the atomic transaction and then either throws an exception or sets an error code. However, this approach would forgo the benefits of compile-time checking and instead shift the burden of detecting and handling atomic transactions that executed unsafe operations to a programmer.

Transaction declaration statements. The features described in this specification do not allow executing an initialization statement inside a transaction without changing the scope of the initialized object (Section 5). We could introduce a transaction declaration statement that causes all the actions initiated by the initialization statement to be performed inside a transaction. A transaction declaration statement would be specified by placing the `__transaction_relaxed` or the `__transaction_atomic` keyword before the declaration as illustrated by the following example, where both the copy constructor and evaluation of its argument are executed within a transaction:

```
__transaction_relaxed SomeObj myObj = expr;
__transaction_atomic SomeObj myObj = expr;
```

Relaxing the lexical scope restriction. We could remove the lexical scoping restriction on cancel statements without `outer` attribute so that such statements could appear anywhere inside the dynamic scope of an atomic transaction. Rollbacks don't make sense outside of the dynamic scope of an atomic transaction, however, so we could define such cancel statements such that they are either a runtime or compile-time error. In the former case, we could define cancel statements executed outside the dynamic scope of an atomic transaction as leading to a runtime failure that terminates the program (similar to a re-throw outside of the dynamic scope of a catch block); for example, by providing a `cancel()` API call that fails if called outside the dynamic scope of an atomic transaction. To support the latter case, we could introduce a new function attribute (e.g., the `transaction_atomic_only` attribute) specifying that a function can only be called within the dynamic extent of an atomic transaction because it may execute a cancel statement outside the lexical scope of an atomic transaction; thus an unannotated `__transaction_cancel` statement must appear within the lexical scope of either an atomic transaction or a properly-declared function (that is, a function declared with the `transaction_atomic_only` or `transaction_may_cancel_outer attribute`). Similar to lexical scoping, this has the advantage that the implementation can distinguish atomic transactions that require rollback. Note, that although an unannotated cancel statement may appear in a function declared with the `transaction_may_cancel_outer` attribute, using a single attribute for functions that may contain an unannotated cancel statement and functions that may contain a cancel-outer statement is not a good idea; such a design decision would artificially restrict the usage of unannotated cancel statements to the dynamic scope of an outer atomic transaction.

Supporting cancelling of relaxed transactions. Allowing cancel statements only inside atomic transactions limits combinations of irrevocable actions and cancel statements to well-structured programming patterns (such as an atomic-within-relaxed idiom in Section 8.3). Alternatively, we could allow arbitrary syntactic combinations of cancel statements and irrevocable actions and place the burden of preventing dynamically unsafe combinations on a programmer. That is, we could allow a cancel statement to appear anywhere within the scope of a relaxed transaction and require that programmers not to use `__transaction_cancel` after a call to an irrevocable action (i.e., any call to an unsafe statement). In this case, cancelling a relaxed transaction that executed an irrevocable action would be a run-time failure that exits the program with an error. We could also devise static rules that avoid rollback after an irrevocable action at the expense of prohibiting some dynamically safe combinations of cancel statements and irrevocable actions.

With this change, we could also forgo differentiating between relaxed and atomic transactions and simply treat relaxed transactions that contain only safe statements as atomic transactions. However, we believe that supporting statically enforced atomic transactions encourages the development of more robust and reliable software by allowing the programmer to declare the intention that a block of code should appear atomic (with the corresponding restriction that it should contain only safe operations). Effectively, atomic transactions act as a compile-time assertion that allows atomicity violations to be identified at compile time rather than run time.

Adding an else clause to atomic transaction statements. We could add an else-clause to "catch" cancels. For example:

```
__transaction_atomic {
    stmt
} else {
        // control ends up here if stmt cancels the transaction
}
```

The else-clause allows the programmer to determine whether an atomic transaction cancelled without resorting to explicit flags. We could also use the else-clause to provide alternate actions in case the atomic transaction attempts to execute an unsafe statement, relaxing the rule that prohibits unsafe function calls inside the dynamic scope of an atomic transaction. Thus, an attempt to execute an unsafe statement inside an atomic transaction would rollback the statement and transfer control to the else-clause.

Introducing a retry statement. We could define a retry statement (e.g., `__transaction_retry`) that rolls back an outer atomic transaction and then re-executes it. Such a retry statement is useful for condition synchronization. Executing a retry statement when the outer atomic transaction is within the dynamic extent of a relaxed transaction, however, will result in an infinite loop (relaxed transactions are serializable with respect to atomic transactions thus re-execution will follow the same path) and may prevent other transactions from making progress (depending on implementation). It might be possible to statically disallow outer atomic transactions from nesting inside a relaxed transaction using additional function attributes, but this might unnecessarily restrict use of code that might execute outer atomic transactions and it introduces a function attribute that might propagate all over the program.

Removing restrictions on types of exceptions thrown by the cancel-and-throw statement. This specification requires exceptions thrown by the cancel-and-throw statement to be of integral or enumerated types. We could remove this restriction and allow the cancel-and-throw statement to throw exceptions of arbitrary types. This, however, could lead to subtle hard-to-detect bugs when an exception object contains or refers to the state that is not meaningful after the transaction is cancelled. For example, if an exception object points to an object allocated inside a transaction, that object would be deallocated when the transaction is cancelled, resulting in a dangling pointer. If an exception object contains a pointer to an object allocated outside of the transaction, throwing this object can still lead to an inconsistent state if the pointer is implemented as a shared pointer with reference count. When transaction is cancelled the increment of the reference count would be undone, possibly causing the thrown object to unexpectedly disappear due the reference count being one too low. Finally, a thrown object may contain inconsistent state even if it contains no pointers. For example, if the thrown object is an instance of a class T, whose constructors and destructors keep track of all instances of T, the tracking of that object is going to be lost after the transaction is cancelled.

Inheriting class attributes. We could let a class with no explicit attribute inherit the class attribute of its base class and define the rules for attribute composition to support multiple inheritance. This would complicate programmer's reasoning while providing a limited benefit of saving one declaration per derived class.

Region attributes. We could introduce region attributes that act as default attributes for functions declared within a region of code. This would allow the programmer to annotate multiple function declarations by specifying the attribute only once. For example, a programmer could annotate all declarations in a header file as `transaction_safe`, by including them in a code region annotated with the `transaction_safe` attribute.

Appendix D. Changes compared to version 1.0

This specification contains the following changes compared to its previous version – the Draft Specification of Transactional Language Constructs for C++, version 1.0:

Transaction keywords. The `__transaction` keyword and its associated attributes, `atomic` and `relaxed`, have been replaced by the `__transaction_atomic` and `__transaction_relaxed` keywords, respectively. Previously, an atomic transaction could be declared by using just the `__transaction` keyword, while a relaxed transaction required the `__transaction` keyword annotated with the `[[relaxed]]` attribute. The new syntax puts relaxed and atomic transactions on equal footing, by providing each with its own keyword.

Transactional types. The transactional function properties defined by `transaction_safe`, `transaction_unsafe`, and `transaction_may_cancel_outer` attributes are now part of a function type. As such, these properties might be specified in typedef declarations and propagated as part of the type. They are still ignored, however, for the overload resolution. Previously, the transactional properties of a function had many characteristics of type without being such, which limited their applicability (e.g., they could not participate in typedef declarations) and left the behavior in multiple corner cases unspecified. Elevating transactional function properties to types solves these problems.

Exception specifications and `noexcept` specifications. The specification now supports C++11's `noexcept` specifications and has removed support for C++11's deprecated exception specifications. This was done because exception specifications have been deprecated in C++11 and have been replaced by `noexcept` specifications.

Cancel-and-throw exception types. The types of exceptions thrown by cancel-and-throw are now limited to integral and enumeration types. This change was made to prevent subtle bugs due to destroyed transactional state escaping the scope of the transaction via an exception object.

Memory model. The memory model now includes complete rules on how TransactionStart and TransactionEnd operations contribute to the "sequenced-before" relationship.

Miscellaneous. The specification contains numerous other minor changes, such as additional examples, fixes to minor inaccuracies and rephrasing of possibly ambiguous statements.