

Draft Specification of Transactional Language Constructs for C++

Version: 1.0

August 4, 2009

Transactional Memory Specification Drafting Group

Editors: Ali-Reza Adl-Tabatabai and Tatiana Shpeisman, 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 subsection (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.

License grant from authors, other terms: The specification is Copyright 2009 by IBM Corporation, Intel Corporation and Sun Microsystems (as joint authors). The authors grant you a license, under their copyright rights in the specification, to reproduce, distribute and create derivative works of the specification, to the limited extent necessary for you to evaluate the specification and to provide feedback to the authors. These legal terms must be included with all copies. The specification is provided "AS IS". Except for the limited license granted in this paragraph, no other license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document.

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
Calin Cascaval, IBM
Steve Clamage, Sun
Robert Geva, Intel
Victor Luchangco, Sun
Virendra Marathe, Sun
Maged Michael, IBM
Mark Moir, Sun
Ravi Narayanaswamy, Intel
Clark Nelson, Intel
Yang Ni, Intel
Daniel Nussbaum, Sun
Tatiana Shpeisman, Intel
Raul Silvera, IBM
Xinmin Tian, Intel
Douglas Walls, Sun
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	5
2. Transaction statement	6
2.1 Memory model	7
3. Atomic transactions	8
3.1 Outer atomic transactions	9
3.2 Safety attributes on functions and function pointers	9
3.3 Examples	11
3.4 Nesting	13
3.5 Memory model	14
4. Relaxed transactions	14
4.1 The <code>transaction_callable</code> function attribute.....	15
4.2 Nesting	15
4.3 Irrevocable actions	15
5. Transaction expressions	15
6. Function transaction blocks	16
7. Exception specifications	17
8. Cancel statement	18
8.1 The <code>outer</code> attribute on cancel statements.....	19
8.2 The <code>transaction_may_cancel_outer</code> attribute on functions and function pointers ...	19
8.3 Examples	20
8.4 Memory model	21
9. Cancel-and-throw statement	22
9.1 The <code>outer</code> attribute on cancel-and-throw statements.....	22
9.2 Examples	23
10. Inheritance and compatibility rules for attributes	24
11. Class attributes	25
Appendix A. Grammar	25
Appendix B. Feature dependences	26
Appendix C. Extensions	29

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 upcoming C++0x 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` keyword (Section 2) can be used to indicate that a compound statement should execute as an atomic transaction; that is, the compound statement 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 compound statement takes effect in its entirety if it takes effect at all. Thus, programmers can think of an atomic transaction statement as taking effect “instantaneously”, which greatly simplifies reasoning about interactions between concurrent threads. Two additional syntactic features allow the programmer to specify expressions (Section 5) and functions (Section 6) that should execute as atomic transactions.

To make this behavior possible, the compiler enforces a restriction that an atomic transaction must contain only “safe” statements (Section 3.2), 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 (Section 3.2). 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 library functions in atomic transactions. Functions may be annotated with the `transaction_unsafe` attribute to prevent such inference. This is useful to avoid the use of a function in atomic transactions 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 3.1). 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 exception specifications (Section 7) to specify which types of exceptions (if any) are expected to be thrown out of the

1 statement; any other type causes a runtime error immediately when the exception escapes the
2 atomic transaction.

3
4 A transaction statement can be annotated with the `relaxed` attribute (Section 4) to relax the
5 rules prohibiting unsafe statements inside transactions, allowing such transaction statements to
6 communicate with other threads and the external world (e.g., via C++0x atomic variables, volatile
7 variables and I/O) while still isolating them from other transactions. Relaxed transaction
8 statements do not allow `cancel` statements and do not guarantee isolation, even in data-race-free
9 programs; that is, concurrent threads with which a transaction communicates (directly or
10 indirectly) can observe partial results of the transaction, and the transaction can observe actions
11 of concurrent threads during the transaction.

12
13 Appendix A includes a grammar for the new features. Appendix B discusses dependences
14 between features, to assist implementers who might be considering implementing subsets of the
15 features described in this document or enabling features in different orders. Appendix C
16 discusses several possible extensions to the features presented in this specification.

17 2. Transaction statement

18 The `__transaction` keyword followed by a compound statement defines a *transaction*
19 *statement*, that is, a statement that executes as a transaction:

20
21 `__transaction compound-statement`

22
23 In a data-race-free program (Section 2.1), all transactions appear to execute sequentially in some
24 total order. This means that transactions execute in isolation from other transactions; that is, the
25 individual steps of a transaction appear not to interleave with individual steps of another
26 transaction.

27
28 [Note: *Although transactions behave as if they execute in some serial order, an implementation*
29 *(i.e., compiler, runtime, and hardware) is free to execute transactions concurrently while providing*
30 *the illusion of serial ordering.*]

31
32 A transaction statement may be optionally annotated with one of the following attributes: `atomic`,
33 `outer` or `relaxed`.¹ A transaction statement with no attribute or with the `atomic` attribute
34 specifies an *atomic transaction* (Section 3). A transaction statement annotated with the `outer`
35 attribute specifies a special kind of an atomic transaction called an *outer atomic transaction*
36 (Section 3.1). A transaction statement annotated with the `relaxed` attribute specifies a *relaxed*
37 *transaction* (Section 4). All of these variants of the transaction statement provide the basic
38 guarantee that transactions appear to execute sequentially in some total order.

39
40 A `goto` or `switch` statement must not be used to transfer control into a transaction statement. A
41 `goto`, `break`, `return`, or `continue` statement may be used to transfer control out of a
42 transaction statement. When this happens, each variable declared in the transaction statement
43 will be destroyed in the context that directly contains its declaration.

44
45 The body of a transaction statement may throw an exception that is not handled inside its body
46 and thus propagates out of the transaction statement (Section 7).

¹ Attributes are specified by declaration in various ways depending on the system. The examples in this document use C++0x attribute syntax – `[[attribute-list]]`. Alternative ways of specifying attributes include GNU syntax – `__attribute__ ((attribute-list))`, and Microsoft syntax – `__declspec (attribute-list)`.

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++0x standard (i.e., locks and C++0x atomic variables). The C++0x 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++0x memory model as defined in Working Draft Doc No: N2857=09-0047:

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++0x 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++0x atomic variables). In addition, there are 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 in the following way:

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 C such that an operation A happens before C and C happens before an operation B then A happens before B.

For programs that do not contain relaxed atomic operations, the above rules completely define the “happens-before” relationship. In the presence of relaxed atomic operations, there may exist an additional “dependency-ordered before” relationship that also contributes to the “happens-before” relationship. (The complete definition of the “happens-before” relationship in C++0x is intricate and is beyond the scope of a brief overview.) 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 race-free if none of its executions contain a data race. In a race-free program each read from a memory location sees the value written by the last write ordered before it by the “happens-before” relationship. Consequently, a race-free program that does not contain C++0x relaxed atomic operations 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.² A StartTransaction operation is sequenced before all other operations of its transaction. All operations of a transaction are sequenced before its EndTransaction operation.

There exists a total order over all StartTransaction and EndTransaction operations called the *transactional synchronization order*. In this order, the individual operations of the transactions executed by different threads do not interleave. In other words, transactional synchronization order is such that a StartTransaction operation executed by one thread does not occur in between a matching pair of StartTransaction and EndTransaction operations executed by another thread. The transactional synchronization order contributes to the “synchronizes with” relationship defined in the C++0x standard. In particular, each EndTransaction operation synchronizes with the next StartTransaction operation in the transactional synchronization order executed by a different thread.

² We introduce these operations purely for the purpose of describing how transaction contribute to the “synchronizes with” relationship.

1 [Note: The definition of the “synchronizes with” relation affects all other parts of the memory
2 model, including the definition of the “happens before” relationship, visibility rules that specify
3 what values can be seen by the reads, and the definition of data race freedom. Consequently,
4 including transactions in the “synchronizes with” relation is the only change to the memory model
5 that is necessary to account for transaction statements. With this extension, the C++0x memory
6 model fully describes the behavior of programs with transaction statements.]
7

8 [Note: A shared memory access can form a data race even if it is performed in a transaction
9 statement. In the following example, a write by thread T2 forms a data race with both read and
10 write to x by Thread T1 because it is not ordered with the operations of Thread T1 by the
11 “happens-before” relationship. To avoid a data race in this example, a programmer should
12 enclose the write to x in Thread T2 in a transaction statement.
13

Thread T1	Thread T2
<pre>__transaction { t = x; x = t+1; }</pre>	<pre>x = 1;</pre>

14]
15
16

17 3. Atomic transactions

18 A transaction statement without an attribute or annotated with the `atomic` attribute defines an
19 *atomic transaction*. We call such a statement an *atomic transaction statement*:

20
21 `__transaction compound-statement`
22 `__transaction [[atomic]] compound-statement`
23

24 In a data-race-free program, an atomic transaction appears to execute atomically; that is, the
25 compound statement appears to execute as a single indivisible operation whose steps do not
26 interleave with the operations of other threads (Section 3.5). Atomic transactions thus allow a
27 programmer to write code fragments that should execute in isolation from other threads (the
28 fragment does not observe changes made by other threads during its execution, and other
29 threads do not observe its partial results before it completes).
30

31 An atomic transaction executes in an all-or-nothing fashion: The operations inside an atomic
32 transaction can be rolled back explicitly via a `cancel` statement (Section 8).
33

34 These properties make it considerably easier to reason about the interactions of the atomic
35 transaction and the actions of other threads.
36

37 To ensure that these guarantees can be made, atomic transactions are statically restricted to
38 contain only “safe” code (Section 3.2). This ensures that an atomic transaction cannot execute
39 code that would have visible side effects before the atomic transaction completes, such as
40 performing synchronization operations and certain I/O. These same restrictions support the ability
41 to roll back an atomic transaction explicitly by executing a `cancel` statement (Section 8), because
42 they ensure that no visible side effects occur during the execution of the atomic transaction, and
43 thus it is possible to roll back all changes made by an atomic transaction at any point during its
44 execution.

3.1 Outer atomic transactions

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

```
__transaction [[ 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 3.2 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.

3.2 Safety attributes on functions and function pointers

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, function declarations (including virtual and template function declarations) may specify `transaction_safe` or `transaction_unsafe` attributes, and function pointer declarations may specify the `transaction_safe` attribute. The `transaction_safe` attribute specifies that a function (or function pointer) may be called within the dynamic extent of an atomic transaction. The `transaction_unsafe` attribute specifies that a function must not be called within the dynamic extent of an atomic transaction. Unsafe statements, defined precisely below, must not appear within atomic transactions or within the bodies of functions declared with the `transaction_safe` attribute.

A function must not be declared with both the `transaction_safe` and `transaction_unsafe` attributes. That is, a function declaration or multiple declarations of one function must not specify both attributes. If any declaration of a function specifies the `transaction_safe` attribute then every declaration of that function (except its definition, if it is not a virtual function) must specify the `transaction_safe` attribute.

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

1. It is a transaction statement annotated with either the `outer` or `relaxed` attribute (that is, it is an outer atomic transaction or a relaxed transaction).
2. It contains an initialization of, assignment to, or a read from a volatile object.
3. It is an unsafe `asm` statement; the definition of the unsafe `asm` statement is implementation-defined.
4. It contains an unsafe function call, as defined below.

[Note: A *relaxed transaction* is *unsafe* because it may contain *unsafe statements* (Section 4). 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` statement 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.)

1
2 A function call is unsafe unless the function was *declared safe* (before the function call), either by
3 being declared with the `transaction_safe` or `transaction_may_cancel_outer` attribute,³
4 or by being implicitly declared safe by the function's definition, as defined below. A function call
5 through a function pointer is *unsafe* unless the function pointer was declared with the
6 `transaction_safe` attribute or the `transaction_may_cancel_outer` attribute.
7

8 A function definition *implicitly declares a function safe* if the function is not a virtual function, its
9 body contains only safe statements, and neither the definition nor any prior declaration of the
10 function specifies any of the `transaction_unsafe`, `transaction_safe`, or
11 `transaction_may_cancel_outer` attributes. (If the definition or a prior declaration specifies
12 the `transaction_safe` or `transaction_may_cancel_outer` attribute, then subsequent
13 calls to that function are safe, but the definition does not implicitly declare the function safe.)
14 If the definition of a function implicitly declares it safe then no declaration of that function may
15 specify the `transaction_unsafe` attribute.
16

17 A function template that does not specify any of the `transaction_safe`,
18 `transaction_unsafe`, or `transaction_may_cancel_outer` attributes may define a
19 template function that may or may not be implicitly declared safe, depending on whether the body
20 of the template function contains unsafe statements after the instantiation. (This feature is
21 especially useful for template libraries, because it allows the use of template library functions
22 within atomic transactions when they are instantiated to contain only safe statements, without
23 requiring these template library functions to be always instantiated to contain only safe
24 statements.) See Section 3.3 for an example of such a function template.
25

26 See Section 10 for rules and restrictions on overriding virtual functions declared with the
27 `transaction_safe` attribute.
28

29 When a function pointer declared with the `transaction_safe` attribute is assigned or initialized
30 with a value, that value must be a function pointer declared with the `transaction_safe`
31 attribute or the address of a function previously declared with the `transaction_safe` attribute
32 or previously implicitly declared safe by its definition.
33

34 [Note: *An implementation may provide additional mechanisms that make statements safe. Such*
35 *mechanisms might be necessary to implement system libraries that execute efficiently inside*
36 *atomic transactions. Such mechanisms are intended for system library developers and are not*
37 *part of this specification.*]
38

39 The creation (destruction) of an object implicitly invokes a constructor (destructor) function if the
40 object belongs to a class that defines a constructor (destructor). The constructor and destructor
41 functions of a class must therefore be declared safe if the programmer intends to allow objects of
42 that class to be created or destroyed inside atomic transactions. In the absence of appropriate
43 programmer-defined constructors (destructors), the creation (destruction) of an object may
44 implicitly invoke a compiler-generated constructor (destructor). Calling a compiler-generated
45 constructor (destructor) is safe if calling the corresponding constructors (destructors) for the class
46 members of the class type and the corresponding constructor (destructor) for the base class is
47 safe. (For example, calling a compiler-generated default constructor is safe if calling the default
48 constructors for the class members of the class type and the default constructor for the base
49 class is safe.) Calling a compiler-generated constructor (destructor) for a class that is not derived
50 from any other class and has no members of class type is always safe.

³ The `transaction_may_cancel_outer` attribute specifies that a function may execute a cancel-
outer statement (Section 8.1), and thus must be called only within the dynamic extent of an outer atomic
transaction. See Section 8.2 for details on the meaning and use of the
`transaction_may_cancel_outer` attribute.

1
2 The assignment to an object invokes a compiler-generated assignment operator if the object
3 belongs to a class that does not define an assignment operator. Calling a compiler-generated
4 assignment operator is safe if calling the assignment operators for the class members of the class
5 type is safe and calling the assignment operator for the base class is safe.

6
7 The `transaction_safe` attribute on function and function pointer declarations allows the
8 compiler to ensure that functions whose bodies contain unsafe statements are not called inside
9 atomic transactions. Any function with external linkage that the programmer intends to be called
10 inside atomic transactions in other translation units must be declared with the
11 `transaction_safe` attribute. To allow client code to use libraries inside atomic transactions,
12 library developers should identify functions with external linkage that are known and intended to
13 contain only safe statements and annotate their declarations in header files with the
14 `transaction_safe` attribute. Similarly, library developers should use the
15 `transaction_unsafe` attribute on functions known or intended to contain unsafe statements.
16 The `transaction_unsafe` attribute specifies explicitly in a function's interface that the function
17 may contain unsafe actions and prevents a function from being implicitly declared safe so that
18 future implementations of that function can contain unsafe statements. When annotating a
19 function with the `transaction_unsafe` attribute, library developers should specify this attribute
20 on both a function declaration and its definition when the declaration and the definition are
21 located in separate header files. This enables client code to include such header files in an
22 arbitrary order.

23
24 *[Note: Library users should not circumvent the restrictions imposed by the library interface by*
25 *merely modifying transaction-related attributes in the library header files. Similar to other changes*
26 *to a function declaration (such as changing a function return type or type of a function argument),*
27 *adding, removing or modifying a transaction-related attribute requires re-compilation. Modifying*
28 *transaction-related attributes in library header files without re-compiling the library may result in*
29 *undefined behavior.]*

30
31 The header files for the C++ standard library should be modified to specify the annotations for the
32 library functions consistent with the safety properties of those functions. Synchronization (that is,
33 operations on locks and C++0x atomic operations) and certain I/O functions in the C++ standard
34 library should be declared unsafe to avoid possible deadlock due to atomic execution of
35 operations that logically cannot be executed atomically.

37 **3.3 Examples**

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

```
39  
40 [[transaction_safe]] void f();
```

41
42 The following example shows a function implicitly declared safe by its definition:

```
43  
44 int x;  
45 void g()  
46 {  
47     x++; // body containing only safe statements  
48 }  
49 // g() implicitly declared safe after this point
```

50
51 An atomic transaction can contain calls to functions declared safe either implicitly or by using an
52 attribute, as illustrated by the following example:

```

1 void test()
2 {
3     __transaction {
4         g(); // OK because g() is implicitly declared safe
5         f(); // OK because f() is declared safe using
6             // the transaction_safe attribute
7     }
8 }
9

```

10 The following example illustrates combinations of declarations:

```

11
12 [[transaction_safe]] int f(); // first declaration of f
13 void f() { x++; } // OK: transaction_safe annotation optional on definition
14 void f(); // Error: prior declaration has transaction_safe attribute
15
16 void g(); // first declaration of g
17 [[transaction_safe]] void g() {...} // Error: prior declaration has no
18 // transaction_safe attribute
19
20 void h() {y++;} // OK, first declaration of h is a definition that implicitly declares it safe
21 [[transaction_unsafe]] void h(); // Error: previous declaration of h
22 // implicitly declared it safe
23
24 [[transaction_unsafe]] void k(); // first declaration of k
25 void k() {z++;}; // OK, this definition does not implicitly declare k safe because of
26 // a prior declaration with the transaction_unsafe attribute
27
28 void l(); // first declaration of l
29 [[transaction_unsafe]] void l(); // OK, first declaration of l did not declare it safe
30
31 void m(); // first declaration of m
32 void m() {w++;} // OK, definition of m implicitly declares it safe
33 void m(); // OK, m is still declared safe
34

```

35 The following example illustrates function pointers declared with the `transaction_safe` attribute:

```

36
37
38 [[transaction_safe]] void (*p1)(int&);
39 void (*p2)(int&);
40 void foo(int&);
41
42 p2 = p1; // OK
43 p2 = f; // OK, f is declared safe
44 p1 = p2; // Error: p2 is not declared with the transaction_safe attribute
45 p1 = foo; // Error: foo is not declared safe
46

```

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

```

48
49
50 template<class Op>
51 void t(int& x, Op f) { // Safety of t is not known at this point
52     x++; f(x);
53 }
54

```

```

1  class A1 {
2  public:
3      [[transaction_safe]] void operator()(int& x); // A1::() is declared safe
4  };
5
6  class A2 {
7  public:
8      [[transaction_unsafe]] void operator()(int& x); // A2::() is declared
9                                     // unsafe
10 };
11
12 void n(int v) {
13     __transaction {
14         t(v, A1()); // OK, call to t<A1> is safe
15         t(v, A2()); // Error, call to t<A2> is unsafe
16     }
17 }

```

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

```

21
22 [[transaction_safe]] void (*p1) (int&);
23 void (*p2) (int&);
24 [[transaction_unsafe]] void u();
25
26 void n(int v) {
27     int total = 0;
28     __transaction {
29         t(v, p1); // OK, the call is safe
30         t(v, [&](int x) {total += x;}); // OK, the call is safe
31         t(v, p2); // Error, the call is unsafe
32         t(v, [&](int x) {u();}); // Error, the call is unsafe
33     }
34 }

```

35 **3.4 Nesting**

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

39
40 The following example shows an atomic transaction lexically nested inside another atomic
41 transaction:

```

42
43 __transaction {
44     x++;
45     __transaction {
46         y++;
47     }
48     z++;
49 }

```

50
51 The following example shows an atomic transaction dynamically nested inside another atomic
52 transaction:

53

```

1  [[ transaction_safe ]] void bar()
2  {
3      __transaction { x++; }
4  }
5
6  __transaction {
7      bar();
8  }

```

3.5 Memory model

The memory model rules for transactions (Section 2.1) are sufficient to guarantee that in race-free programs, atomic transactions appear to execute as single indivisible operations. Atomic transactions cannot contain other forms of synchronization (such as operations on locks or C++0x atomic operations). Consequently, an operation executed by one thread cannot be ordered by the “happens-before” relationship in 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.

4. Relaxed transactions

A transaction statement annotated with the `relaxed` attribute defines a *relaxed transaction*. We call such a statement a *relaxed transaction statement*.

```
__transaction [[ relaxed ]] compound-statement
```

Unlike atomic transactions, relaxed transactions may contain unsafe statements. Relaxed transactions that execute unsafe statements may appear to interleave with non-transactional operations from other threads (or I/O from the external world); therefore, unlike atomic transactions, relaxed transactions do not necessarily appear to execute as single indivisible operations. Relaxed transactions that execute only safe statements behave the same way as atomic transactions; that is, they appear to execute as single indivisible operations.

The following example illustrates a valid thread interleaving of a race-free program in which a relaxed transaction executes non-atomically. 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++0x atomic variables cannot create a data race (Section 3.5):

Initially <code>atomic<int> x = 0;</code>	
Thread T1	Thread T2
<pre> __transaction [[relaxed]] { x = 1; while (x != 0) {} } </pre>	<pre> while (x != 1) {} x = 0; </pre>

Because the relaxed transactions allow interleaving of operations from other threads, they do not necessarily force a deadlock for code that will deadlock if executed atomically; for example, code that communicates with other threads (such as the previous code sequence) or with the external world and waits for a response need not deadlock when executed inside a relaxed transaction. (However, such code will deadlock if the operations of the other thread are themselves enclosed in a transaction as operations within different transactions cannot interleave.)

1 Unlike atomic transactions, relaxed transactions support the interoperability of transactions with
2 synchronization operations and I/O. Relaxed transactions provide the ability to interact with
3 existing synchronization primitives and to use them to escape isolation guarantees of atomic
4 transactions. They also provide the ability to execute operations that cannot be rolled back, such
5 as I/O and other system calls. Relaxed transactions are thus useful as a tool to incrementally port
6 existing applications that use different synchronization primitives.

7 **4.1 The `transaction_callable` function attribute**

8 The `transaction_callable` attribute indicates that a function (including virtual functions and
9 template functions) is intended to be called within a relaxed transaction. The
10 `transaction_callable` attribute is intended for use by an implementation to improve the
11 performance of relaxed transactions; for example, an implementation can generate a specialized
12 version of a `transaction_callable` function, and execute that version when the function is
13 called inside a relaxed transaction. A function need not be declared with the
14 `transaction_callable` attribute to be called inside a relaxed transaction.

15
16 The `transaction_callable` attribute does not affect the safety of the declared function. A
17 function declared with the `transaction_callable` attribute may also be declared with the
18 `transaction_safe`, `transaction_unsafe` or `transaction_may_cancel_outer`
19 attribute, or be implicitly declared safe. A function declared with the `transaction_callable`
20 attribute must not be redeclared without that attribute. A function declared without the
21 `transaction_callable` attribute must not be redeclared with the `transaction_callable`
22 attribute.

23
24 See Section 10 for rules and restrictions on overriding virtual functions declared with the
25 `transaction_callable` attribute.
26

27 **4.2 Nesting**

28 Atomic transactions may be nested within relaxed transactions. Relaxed transactions may be
29 nested within other relaxed transactions. Relaxed transactions must not be nested within atomic
30 transactions (that is, they are unsafe).

31 **4.3 Irrevocable actions**

32 Unlike an atomic transaction, a relaxed transaction may execute unsafe statements. Some
33 unsafe statements may have side effects that the system cannot roll back. We refer to such
34 operations as *irrevocable* actions. For example, communicating partial results of a relaxed
35 transaction to either the external world via an I/O operation or to other threads via a
36 synchronization operation (such as a lock release or a write to a C++0x atomic variable) may
37 constitute an irrevocable action because the system may not be able to roll back the effects that
38 this communication had on the external world or other threads.

39
40 Irrevocable actions may limit the concurrency in an implementation; for example, they may cause
41 the implementation to not execute relaxed transactions concurrently with other transactions.
42

43 **5. Transaction expressions**

44 The `__transaction` keyword followed by a parenthesized expression defines a *transaction*
45 *expression*. A transaction expression may be optionally annotated with either the `atomic` or
46 `relaxed` attribute. Unlike a transaction statement, a transaction expression must not be
47 annotated with the `outer` attribute:
48

```
1      __transaction ( expression )
2      __transaction [[ atomic ]] ( expression )
3      __transaction [[ relaxed ]] ( 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:

```
8      __transaction { 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:

```
16     SomeObj myObj = __transaction ( 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:

```
42     void f() __transaction {
43         // body of f() executes in a transaction
44     }
```

Like a transaction expression, a function transaction block may be optionally annotated with either the `atomic` or `relaxed` attribute, but not with the `outer` attribute:

```
49     void f() __transaction [[ atomic ]] { ... }
50     void f() __transaction [[ relaxed ]] { ... }
```

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

1 block syntax thus allows programmers to include member and base class initializers in
2 constructors in a transaction. In the following example, the constructor `Derived()` and its
3 initializers all execute atomically:

```
4  
5 class Base {  
6     const int id;  
7     Base(int _id) : id(_id) {}  
8 };  
9  
10 class Derived : Base {  
11     static int count = 0;  
12     Derived() __transaction : Base(count++) { ... }  
13 };  
14
```

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

21
22 A function transaction block can be combined with the function try block syntax. If the
23 `__transaction` keyword appears before the `try` keyword, the catch block is part of the
24 function transaction block. If the `__transaction` keyword appears after the `try` keyword, the
25 catch block is not part of the function transaction block:

```
26  
27 Derived::Derived()  
28 try __transaction : Base(count++) {}  
29 catch (...) {} // catch is not part of transaction  
30 Derived::Derived()  
31 __transaction try : Base(count++) { ... }  
32 catch (...) {} // catch is part of transaction  
33
```

34 Unlike a transaction statement, a function transaction block may contain a `cancel` statement only
35 if that `cancel` statement is annotated with the `outer` attribute (Section 8). A function transaction
36 block may contain a `cancel-and-throw` statement (Section 9).

37 7. Exception specifications

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

40
41 Transaction statements and expressions may have exception specifications that explicitly list the
42 exception types that the statement may throw:

```
43  
44 __transaction throw( type-id-list ) compound-statement  
45 __transaction throw( type-id-list ) ( expression )  
46
```

47 Transaction statements and expressions that have exception specifications may be annotated
48 with an attribute, which should appear in between the `__transaction` keyword and the
49 exception specification, as illustrated by the following example:

```
50  
51 __transaction [[ relaxed ]] throw( type-id-list ) compound-statement  
52
```

1 The type of an exception thrown by a transaction statement (expression) must be listed in the
2 exception specification, or derived from a type listed in the exception specification. An exception
3 of any other type results in a call to `std::unexpected()`. An exception specification that lists
4 no types (i.e., `throw()`) declares that the statement (expression) throws no exceptions. The
5 following example illustrates the use of an exception specification to declare that a transaction
6 statement may throw exceptions of type `X` or `Y`:

```
7  
8 __transaction throw(X, Y) compound-statement
```

9
10 The following example similarly declares that the transaction statement throws no exceptions:

```
11  
12 __transaction throw() compound-statement
```

13
14 The programmer can specify that a transaction statement (expression) can throw an exception of
15 any type using an ellipsis as the exception list:

```
16  
17 __transaction throw(...) compound-statement
```

18
19 A transaction statement (expression) without an exception specification is equivalent to that
20 statement (expression) with a `throw(...)` clause; that is, a transaction statement or expression
21 without an exception specification may throw any exception.

22
23 [Note: *Omitting an exception specification on a transaction statement (expression) that may throw*
24 *an exception makes it easy to overlook the possibility that an exception thrown from within the*
25 *dynamic extent of that statement (expression) can result in the statement (expression) being only*
26 *partially executed. Therefore, programmers are strongly encouraged to use explicit exception*
27 *specifications. We considered an alternative approach in which the absence of an exception*
28 *specification is interpreted as if a `throw()` clause were present, which makes mandatory an*
29 *explicit exception specification on a transaction statement (expression) that may throw an*
30 *exception. However, such an interpretation would be inconsistent with the existing rules for*
31 *exception specifications on function declarations.]*

32
33 An exception specification is not allowed on a function transaction block as such a specification is
34 redundant with an exception specification on a function declaration (that is, an exception
35 specification that may appear before the `__transaction` keyword denoting a function
36 transaction block).

37 8. Cancel statement

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

```
42  
43 __transaction {  
44     stmt1  
45     __transaction_cancel;  
46 }  
47 stmt2
```

48
49 In its basic form (that is, without the `outer` attribute), a cancel statement rolls back all side
50 effects of the immediately enclosing atomic transaction statement (that is, the smallest atomic
51 transaction statement that encloses the cancel statement) and transfers control to the statement
52 following the transaction statement. Thus, in the example above the cancel statement undoes the
53 side effects of `stmt1` and transfers control to `stmt2`.

1
2 The rule requiring a cancel statement to be lexically enclosed in an atomic transaction statement
3 ensures that the cancel statement always executes within the dynamic extent of an atomic
4 transaction statement. It also allows the implementation to distinguish easily between atomic
5 transactions that require rollback and those that don't, a potential optimization opportunity for the
6 implementation.

7
8 [Note: A cancel statement applies only to atomic transaction statements (including outer atomic
9 transaction statements). A cancel statement cannot be used to roll back a function transaction
10 block or a transaction expression, unless that block or expression is rolled back as part of rolling
11 back an atomic transaction statement.]

12 **8.1 The outer attribute on cancel statements**

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

```
14  
15 __transaction_cancel [[ outer ]];
```

16
17 We call a cancel statement with the `outer` attribute a *cancel-outer* statement. A cancel-outer
18 statement rolls back all side effects of the outer atomic transaction that dynamically contains it
19 (which is also the outermost atomic transaction that dynamically contains it) and transfers control
20 to the statement following the outer atomic transaction.

21
22 Unlike a cancel statement with no attribute, a cancel-outer statement need not be enclosed within
23 the lexical scope of an atomic transaction. Instead, to ensure that a cancel-outer statement
24 always executes within the dynamic extent of an outer atomic transaction, a cancel-outer
25 statement must appear either within the lexical scope of an outer atomic transaction or in a
26 function declared with the `transaction_may_cancel_outer` attribute (Section 8.2).

27
28 [Note: A cancel-outer statement cancels only outer atomic transactions; the restrictions above
29 imply that a cancel-outer statement cannot be executed when the outermost atomic transaction is
30 not an outer atomic transaction. In contrast, an unannotated cancel statement can cancel an
31 outer atomic transaction if it is the immediately enclosing atomic transaction.]

32
33 The cancel-outer statement provides a convenient way to cancel an outermost atomic transaction
34 from anywhere within its dynamic extent; for example, on encountering an error condition. The
35 outer atomic transaction – together with the `transaction_may_cancel_outer` attribute –
36 ensures that an outermost atomic transaction that may dynamically contain a cancel-outer
37 statement is easily identifiable as such. This is important because otherwise, it would be difficult
38 to determine whether a given atomic transaction might be cancelled without examining all code it
39 might call.

40
41 [Note: Cancelling an outermost atomic transaction using either multiple cancel statements without
42 the `outer` attribute or exceptions both have the disadvantages that additional, error-prone code
43 would be required to transfer control back to the outermost atomic transaction and to cancel the
44 outermost atomic transaction.]

45 **8.2 The `transaction_may_cancel_outer` attribute on** 46 **functions and function pointers**

47
48 Function declarations (including virtual and template function declarations) and function pointer
49 declarations may specify the `transaction_may_cancel_outer` attribute. The
50 `transaction_may_cancel_outer` attribute specifies that the declared function (or a function
51 pointed to by the declared function pointer) may contain a cancel-outer statement within its

1 dynamic scope. We call a function declared with the `transaction_may_cancel_outer`
2 attribute a *cancel-outer function*. Like cancel-outer statements, a call to a cancel-outer function
3 must appear either within the lexical scope of an outer atomic transaction or in a cancel-outer
4 function.

5
6 A function or function pointer must not be declared with both the
7 `transaction_may_cancel_outer` and `transaction_safe` attributes. A function must not
8 be declared with both the `transaction_may_cancel_outer` and `transaction_unsafe`
9 attributes. That is, a function declaration, a function pointer declaration, or multiple declarations of
10 one function must not specify both attributes.⁴ If any declaration of a function specifies the
11 `transaction_may_cancel_outer` attribute then every declaration of that function (except its
12 definition, if it is not a virtual function) must specify the `transaction_may_cancel_outer`
13 attribute, and the first declaration must do so even if it is the definition of a non-virtual function.
14 The main function must not be declared with the `transaction_may_cancel_outer` attribute.

15
16 A function declaration that specifies the `transaction_may_cancel_outer` attribute declares
17 that function safe (Section 3.2). That is, a function call to a function declared with the
18 `transaction_may_cancel_outer` attribute (before the function call) is a safe statement. A
19 function call through a function pointer that was declared with the
20 `transaction_may_cancel_outer` attribute is also a safe statement. The body of a function
21 declared with the `transaction_may_cancel_outer` attribute must not contain unsafe
22 statements.

23
24 See Section 10 for rules and restrictions on overriding virtual functions declared with the
25 `transaction_may_cancel_outer` attribute.

26
27 When a function pointer declared with the `transaction_may_cancel_outer` attribute is
28 assigned or initialized with a value, that value must be a function pointer declared with the
29 `transaction_may_cancel_outer` attribute or a value that may be assigned to a function
30 pointer declared with the `transaction_safe` attribute (that is, a function pointer declared with
31 the `transaction_safe` attribute or the address of a function previously declared safe).
32 When a function pointer declared without the `transaction_may_cancel_outer` attribute is
33 assigned or initialized with a value, that value must not be a function pointer declared with the
34 `transaction_may_cancel_outer` attribute or an address of a function declared with the
35 `transaction_may_cancel_outer` attribute.

36 **8.3 Examples**

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

```
39  
40 bool flag1 = false, flag2 = false;  
41 __transaction {  
42     flag1 = true; // stmt1  
43     __transaction {  
44         flag2 = true; // stmt2  
45         __transaction_cancel;  
46     }  
47     assert (flag1 == true && flag2 == false);  
48 }  
49 assert (flag1 == true && flag2 == false);  
50
```

⁴ Function pointer declarations may not specify the `transaction_unsafe` attribute (Section 3.2).

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

```

4
5 bool flag1 = false, flag2 = false;
6 __transaction [[outer]] {
7     flag1 = true; // stmt1
8     __transaction {
9         flag2 = true; // stmt2
10        __transaction_cancel [[outer]];
11    }
12    assert (0); // never reached!
13 }
14 assert (flag1 == false && flag2 == false);
15 
```

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

```

19
20 [[transaction_safe]] void do_work();
21 [[transaction_safe]] bool all_is_ok();
22 [[transaction_unsafe]] void report_results(); // contains irrevocable actions
23
24 __transaction [[ relaxed ]] {
25     bool all_ok = false;
26     __transaction {
27         do_work();
28         if (all_is_ok())
29             all_ok = true;
30         else
31             __transaction_cancel;
32     }
33     if (all_ok)
34         report_results();
35 }

```

36 **8.4 Memory model**

37 Cancelling an atomic transaction removes all side effects of its execution. Consequently, in a
 38 race-free program a cancelled atomic transaction has no visible side effects. Cancelling an
 39 atomic transaction, however, does not remove a data race that occurred during the execution of
 40 the transaction. The individual operations of an atomic transaction that executed before the
 41 transaction was cancelled are part of the program execution and, like other operations, may
 42 contribute to data races. In case of a data race, the program behavior is still undefined, as
 43 specified by the C++0x memory model. For example, the following program is deemed racy even
 44 though the transaction with a racy memory access is cancelled:
 45

Thread 1	Thread 2
<pre> __transaction { x++; __transaction_cancel; } </pre>	<pre> x = 1; </pre>

46

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 throw (exception-list) {  
    stmt1  
    __transaction_cancel throw throw-expression;  
}
```

The cancel-and-throw statement first copies the specified exception into a special exception area in memory. It then rolls back all side effects of the immediately enclosing atomic transaction statement (atomic function transaction block) (i.e., it undoes the side effects of *stmt1* in the above example), but this rollback does not undo the values written to the special exception area by the copy. Finally, the copied exception is thrown from the transaction.

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

In a catch clause, 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 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.

If the exception thrown by a cancel-and-throw statement was allocated or modified within the transaction, then it might contain or refer to state that will not be meaningful after the transaction is cancelled. It is the programmer's responsibility to ensure that the exception object is meaningful after the transaction is cancelled. For example, the programmer must take care to ensure that in the exception's copy constructor, the thrown exception does not point to memory that is rolled back (including memory allocated by the copy constructor). The programmer must be careful not to overlook the fact that an object whose pointer is copied by a copy constructor may be destroyed as a result of an atomic transaction being cancelled, resulting in a dangling pointer.

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 expropt;
```

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

1 and throws the exception from the outer atomic transaction. Like the cancel-outer statement, a
2 cancel-outer-and-throw statement need not be enclosed within the lexical scope of an atomic
3 transaction, but it must appear either within the lexical scope of an outer atomic transaction or in
4 a cancel-outer function.

5 **9.2 Examples**

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

```
10  
11 bool flag1 = false, flag2 = false;  
12 try {  
13     __transaction throw(X) {  
14         flag1 = true; // stmt1  
15         __transaction throw(X) {  
16             flag2 = true; // stmt2  
17             __transaction_cancel throw X();  
18         }  
19     }  
20 } catch(X& x) {  
21     assert(flag1 == true && flag2 == false);  
22 }  
23
```

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

```
28  
29 bool flag1 = false, flag2 = false;  
30 try {  
31     __transaction [[outer]] throw(X) {  
32         flag1 = true; // stmt1  
33         __transaction throw(X) {  
34             flag2 = true; // stmt2  
35             __transaction_cancel [[outer]] throw X();  
36         }  
37     }  
38 } catch(X& x) {  
39     assert(flag1 == false && flag2 == false);  
40 }  
41
```

42 The exception thrown by a cancel-and-throw statement cannot be caught by any try-catch block
43 nested within the cancelled atomic transaction. In the following example, the first catch block
44 does not catch the exception thrown by the cancel-and-throw:
45

```

1  try {
2      __transaction throw(X) {
3          try {
4              __transaction_cancel throw X();
5          } catch(X& x) {
6              assert(0); // never reached!
7          }
8      }
9  } catch (X& x) {
10     cout << "Caught X!" << endl;
11 }

```

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:

```

16 __transaction throw(...) {
17     try {
18         stmt
19     } catch (...) {
20         __transaction_cancel throw;
21     }
22 }
23
24

```

25 10. Inheritance and compatibility rules for attributes

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

- 33 • A virtual function declared with the `transaction_safe` attribute may be overridden
34 only by a virtual function declared with the `transaction_safe` attribute.
- 35 • A virtual function declared with the `transaction_may_cancel_outer` attribute can be
36 overridden only by a virtual function declared with either the
37 `transaction_may_cancel_outer` attribute or `transaction_safe` attribute.
- 38 • A virtual function declared with the `transaction_may_cancel_outer` attribute must
39 not override a virtual function not declared with the `transaction_may_cancel_outer`
40 attribute.

41
42 The following example illustrates the class inheritance rules for transaction-related function
43 attributes:

```

44
45 class C {
46     [[transaction_safe]] void f();
47     [[transaction_safe]] virtual void v();
48     [[transaction_unsafe]] virtual void w();
49 };

```

```

1 class D : public C {
2     void f(); // OK: D::f redefines C::f
3     virtual void v(int); // Error: D::v overrides C::v; needs transaction_safe
4     virtual void w(int); // OK: transaction_unsafe on D::w is optional
5     using C::v; // OK: C::v preserves the transaction_safe attribute
6 };
7
8

```

9 11. Class attributes

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

18
19 The following example shows a definition of class C from Section 10 written using class
20 attributes:

```

21
22 class C [[transaction_safe]] {
23     void f(); // declared as transaction_safe
24     virtual void v(); // declared as transaction_safe
25     [[transaction_unsafe]] virtual void w(); // declared as
26 // transaction_unsafe
27 };
28

```

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

32 Appendix A. Grammar

33 *transaction-statement:*

```
34 __transaction txn-attributeopt txn-exception-specopt compound-statement
```

35

36 *transaction-expression:*

```
37 __transaction txn-base-attributeopt txn-exception-specopt ( expression )
```

38

39 *function-transaction-block:*

```
40 basic-function-transaction-block
```

```
41 __transaction txn-base-attributeopt basic-function-try-block
```

42

43 *basic-function-transaction-block*

```
44 __transaction txn-base-attributeopt ctor-initializeropt compound-statement
```

45

46 *cancel-statement:*

```
47 __transaction_cancel txn-outer-attributeopt i
```

48

```

1  cancel-and-throw-statement:
2      __transaction_cancel txn-outer-attributeopt throw-expression ;
3
4  txn-exception-spec:
5      exception-specification
6      throw ( ... )
7
8  txn-attribute:
9      txn-base-attribute
10     txn-outer-attribute
11
12  txn-base-attribute:
13      [[ atomic ]]
14      [[ relaxed ]]
15
16  txn-outer-attribute:
17      [[ outer ]]
18
19  postfix-expression:
20      /* ... existing C++0x rules ... */
21      transaction-expression
22
23  statement:
24      /* ... existing C++0x rules ... */
25      attribute-specifieropt transaction-statement
26
27  jump-statement:
28      /* ... existing C++0x rules ... */
29      cancel-statement
30      cancel-and-throw-statement
31
32  function-body:
33      /* ... existing C++0x rules ... */
34      function-transaction-block
35
36  function-try-block:
37      basic-function-try-block
38      try basic-function-transaction-block handler-seq
39
40  basic-function-try-block:
41      /* ... existing C++0x rules for function-try-block ... */

```

42 **Appendix B. Feature dependences**

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

46

1 As general guidance, we recommend that an implementation that does not support a certain
2 feature accepts the syntax of that feature and issues an informative error message, preferably
3 indicating that the feature is not supported by the implementation but is a part of the specification.
4

5 The language features described in this specification are interdependent. Eliminating a certain
6 feature may make some other features unusable. For example, without the outer atomic
7 transactions, the cancel-outer statement is unusable; that is, it is not possible to write a legal
8 program that executes a cancel-outer statement and does not contain an outer atomic transaction
9 statement (because the cancel-outer statement must execute within the dynamic extent of an
10 outer atomic transaction). Some other features may remain usable but become irrelevant. For
11 example, without atomic transactions, the `transaction_safe` attribute can occur in legal
12 programs but serves no purpose. We recommend that an implementation that chooses to support
13 a certain irrelevant feature issues an informative warning specifying that the feature is supported
14 for compatibility purposes but has no effect. In the rest of this section, we describe dependences
15 between the features and identify the consequences of omitting a particular feature or
16 combination of features.
17

18 **Transaction statements, transaction expressions and function transaction blocks.** This
19 specification provides three language constructs for specifying transactions: transaction
20 statements, transaction expressions and function transaction blocks. All other features described
21 in this specification are dependent on the presence of at least one of these constructs. Therefore
22 any implementation should include at least one of these constructs. The constructs themselves
23 are independent of each other. An implementation may include one, two or all three of them.
24

25 All three constructs allow for specifying two forms of transactions – atomic transactions and
26 relaxed transactions. Furthermore, atomic statements may be annotated with the `outer` attribute
27 to indicate that they execute as outer atomic transactions. These forms of transactions are
28 independent of each other. An implementation may include either atomic transactions, or relaxed
29 transactions, or both. It may also choose not to support outer atomic transactions, or to require all
30 atomic transactions to be outer atomic transactions.
31

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

- 35 • The concept of safe and unsafe statements and the `transaction_safe` and
36 `transaction_unsafe` function attributes are irrelevant without atomic transactions
37 (because the safety concept and attributes are used to impose restrictions on statements that
38 can be executed within an atomic transaction).
- 39 • The cancel statement is unusable without atomic transaction statements (because it applies
40 only to atomic transaction statements).
- 41 • The cancel-and-throw statement is unusable unless an implementation supports either
42 atomic transaction statements or atomic function transaction blocks (because it applies only
43 to atomic transaction statements or atomic function transaction blocks).
- 44 • The cancel-outer statement, the cancel-outer-and-throw statement, and the
45 `transaction_may_cancel_outer` attribute are unusable without outer atomic
46 transactions (because the cancel-outer statements, cancel-outer-and-throw statements and
47 calls to functions declared with the `transaction_may_cancel_outer` attribute can
48 execute only within the dynamic extent of an outer atomic transaction).
49

50 The only feature used solely in conjunction with the relaxed transactions is the
51 `transaction_callable` attribute. This attribute is irrelevant without relaxed transactions
52 (because it indicates that a function might be called within a relaxed transaction).
53

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

1
2 **Function call safety.** This specification includes three features related to the safety of function
3 calls – the `transaction_safe` and `transaction_unsafe` attributes and the concept of
4 functions being implicitly declared safe. Eliminating one or more of the function call safety
5 features does not affect the rest of the specification. However, different combinations of these
6 features offer different degrees of ability to call functions from within atomic transactions:
7

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

26
27 An implementation may include the `transaction_safe` attribute for function declarations, or
28 function pointer declarations, or both. An implementation that does not support the
29 `transaction_safe` attribute for function pointer declarations must disallow calls via function
30 pointers inside atomic transactions.

31
32 **Cancel and cancel-and-throw statements.** This specification provides two forms of a cancel
33 statement – a basic cancel statement that cancels the immediately enclosing atomic transaction
34 and the cancel-outer statement that cancels the enclosing outer atomic transaction. This
35 specification also provides two similar forms of a cancel-and-throw statement – a basic cancel-
36 and-throw statement and the cancel-and-throw-outer statement. The cancel and cancel-and-
37 throw statements and the two forms of each statement are independent of each other. An
38 implementation may include any combination of these statements and their forms. Eliminating
39 either the basic cancel statement or the basic cancel-and-throw statement does not affect the rest
40 of the specification. Eliminating either the cancel-outer statement or the cancel-outer-and-throw
41 statement, but not both of these statements, also does not affect the rest of the features.
42 Eliminating both the cancel-outer statement and the cancel-outer-and-throw statement makes the
43 `transaction_may_cancel_outer` attribute irrelevant (because this attribute is used to specify
44 that a function may contain either the cancel-outer or cancel-outer-and-throw statement in its
45 dynamic scope) and limits the usability of the `outer` attribute on transaction statements (because
46 the main purpose of this attribute is to specify atomic transactions that can be cancelled by the
47 cancel-outer or cancel-outer-and-throw statement). The `outer` attribute, however, still can be
48 used to specify that an atomic transaction statement cannot be nested within another atomic
49 transaction.

50
51 **The `transaction_may_cancel_outer` attribute.** Eliminating the
52 `transaction_may_cancel_outer` attribute reduces the usability of the cancel-outer and
53 cancel-outer-and-throw statements. An implementation that does not support this attribute must
54 not allow the cancel-outer and the cancel-outer-and-throw statements outside of the lexical scope

1 of an outer atomic transaction statement (because the implementation has no ability to verify that
2 a function containing a cancel-outer statement in its dynamic scope is not called outside of an
3 outer atomic transaction).

4
5 **The `transaction_callable` attribute.** This attribute has no semantic meaning: it is only a
6 hint to the compiler that certain optimizations might be worthwhile. Eliminating this attribute has
7 no effect on other features.

8
9 **Exception specification.** An exception specification facilitates development of more reliable
10 programs. Not supporting exception specifications on transaction statements and/or expressions
11 has no effect on other features.

12
13 **Exceptions.** An implementation that implements a subset of this specification may choose to
14 provide limited support for exceptions inside transactions (including the exceptions thrown by the
15 throw statement and/or exceptions thrown by the cancel-and-throw statement). For example, an
16 implementation might disallow throwing an exception from within code that could be executed
17 within a transaction, or disallow exceptions from escaping the scope of a transaction. Such
18 restrictions might make exception specifications irrelevant.

19
20 **Unsafe statements.** This specification defines certain statements as unsafe. An implementation
21 that implements a subset of this specification might choose to treat additional statements as
22 unsafe. For example, an implementation might choose to treat built-in `new` and `delete` operators
23 as unsafe and disallow them inside atomic transactions. We suggest that such an implementation
24 provides a workaround to allow programmers to allocate and deallocate objects within atomic
25 transactions, and indicate this in an error message produced when encountering a `new` or
26 `delete` built-in operator in an atomic transaction. In most cases, treating additional statements
27 as unsafe would not affect the rest of the specification.

28
29 **Class attributes.** Class attributes have no semantic meaning: they are default attributes for
30 function members declared without a transaction-related attribute. Eliminating class attributes has
31 no effect on the rest of the features.

32 33 **Appendix C. Extensions**

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

41
42 **Transaction declaration statements.** The features described in this specification do not allow
43 executing an initialization statement inside a transaction without changing the scope of the
44 initialized object (Section 5). We could introduce a transaction declaration statement that causes
45 all the actions initiated by the initialization statement to be performed inside a transaction. A
46 transaction declaration statement would be specified by placing the `__transaction` keyword
47 before the declaration as illustrated by the following example, where both the copy constructor
48 and evaluation of its argument are executed within a transaction:

```
49     __transaction SomeObj myObj = expr;
```

50
51
52 **Relaxing the lexical scope restriction.** We could remove the lexical scoping restriction on
53 cancel statements without `outer` attribute so that such statements could appear anywhere inside

1 the dynamic scope of an atomic transaction. Rollbacks don't make sense outside of the dynamic
2 scope of an atomic transaction, however, so we could define such cancel statements such that
3 they are either a runtime or compile-time error. In the former case, we could define cancel
4 statements executed outside the dynamic scope of an atomic transaction as leading to a runtime
5 failure that terminates the program (similar to a re-throw outside of the dynamic scope of a catch
6 block); for example, by providing a `cancel()` API call that fails if called outside the dynamic
7 scope of an atomic transaction. To support the latter case, we could introduce a new function
8 attribute (e.g., the `transaction_atomic_only` attribute) specifying that a function can only be
9 called within the dynamic extent of an atomic transaction because it may execute a cancel
10 statement outside the lexical scope of an atomic transaction; thus an unannotated
11 `__transaction_cancel` statement must appear within the lexical scope of either an atomic
12 transaction or a properly-declared function (that is, a function declared with the
13 `transaction_atomic_only` or `transaction_may_cancel_outer` attribute). Similar to
14 lexical scoping, this has the advantage that the implementation can distinguish atomic
15 transactions that require rollback. Note, that although an unannotated cancel statement may
16 appear in a function declared with the `transaction_may_cancel_outer` attribute, using a
17 single attribute for functions that may contain an unannotated cancel statement and functions that
18 may contain a cancel-outer statement is not a good idea; such a design decision would artificially
19 restrict the usage of unannotated cancel statements to the dynamic scope of an outer atomic
20 transaction.

21
22 **Supporting cancelling of relaxed transactions.** Allowing cancel statements only inside atomic
23 transactions limits combinations of irrevocable actions and cancel statements to well-structured
24 programming patterns (such as an atomic-within-relaxed idiom in Section 8.3). Alternatively, we
25 could allow arbitrary syntactic combinations of cancel statements and irrevocable actions and
26 place the burden of preventing dynamically unsafe combinations on a programmer. That is, we
27 could allow a cancel statement to appear anywhere within the scope of a relaxed transaction and
28 tell programmers not to use `__transaction_cancel` after a call to an irrevocable action (i.e.,
29 any call to an unsafe statement). In this case, canceling a relaxed transaction that executed an
30 irrevocable action would be a run-time failure that exits the program with an error. We could also
31 devise static rules that avoid rollback after an irrevocable action at the expense of prohibiting
32 some dynamically safe combinations of cancel statements and irrevocable actions.

33
34 With this change, we could also forgo differentiating between atomic and relaxed transactions
35 using attributes and simply treat relaxed transactions that contain only safe statements as atomic
36 transactions. However, providing separate atomic transactions encourages the development of
37 more robust and reliable software by allowing the programmer to declare the intention that a
38 block of code should appear atomic (with the corresponding restriction that it should contain only
39 safe operations). Effectively, atomic transactions act as a compile-time assertion that allows
40 atomicity violations to be identified at compile time rather than run time.

41
42 **Adding an else clause to atomic transaction statements.** We could add an else-clause to
43 "catch" cancels. For example:

```
44  
45     __transaction {  
46         stmt  
47     } else {  
48         // control ends up here if stmt cancels the transaction  
49     }
```

50
51 The else-clause allows the programmer to determine whether an atomic transaction cancelled
52 without resorting to explicit flags. We could also use the else-clause to provide alternate actions
53 in case the atomic transaction attempts to execute an unsafe statement, relaxing the rule that
54 prohibits unsafe function calls inside the dynamic scope of an atomic transaction. Thus, an

1 attempt to execute an unsafe statement inside an atomic transaction would rollback the statement
2 and transfer control to the else-clause.
3

4 **Introducing a retry statement.** We could define a retry statement (e.g.,
5 `__transaction_retry`) that rolls back an outer atomic transaction and then re-executes it.
6 Such a retry statement is useful for condition synchronization. Executing a retry statement when
7 the outer atomic transaction is within the dynamic extent of a relaxed transaction, however, will
8 result in an infinite loop (relaxed transactions are serializable with respect to atomic transactions
9 thus re-execution will follow the same path) and may prevent other transactions from making
10 progress (depending on implementation). It might be possible to statically disallow outer atomic
11 transactions from nesting inside a relaxed transaction using additional function attributes, but this
12 might unnecessarily restrict use of code that might execute outer atomic transactions and it
13 introduces a function attribute that might propagate all over the program.
14

15 **Cancel-and-throw-aware allocator.** An exception object thrown by a cancel-and-throw
16 statement may contain a dangling pointer, if that object points to a memory that is destroyed as a
17 result of an atomic transaction being cancelled. We could address the dangling pointer issue by
18 arranging for all allocations performed during a transactional throw operation to return blocks of
19 memory that reside in an area that is not rolled back by the subsequent cancel operation. The
20 “special exception area” introduced in Section 8 may be suitable for that purpose. If such an
21 allocator were available, then the user could solve the dangling-pointer problem with a copy
22 constructor that performs a deep copy.
23

24 Such a “transaction-aware” allocator might either replace the default allocator (called by the
25 default `new` operator), or else we might add a function to answer the question “Am I currently
26 running a cancel-and-throw operation?”, in which case the new allocator could be explicitly
27 (conditionally) called by user code, either from the copy constructor or directly in line, at the throw
28 site.
29

30 **Inheriting class attributes.** We could let a class with no explicit attribute inherit the class
31 attribute of its base class and define the rules for attribute composition to support multiple
32 inheritance. This would complicate programmer’s reasoning while providing a limited benefit of
33 saving one declaration per derived class.
34

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